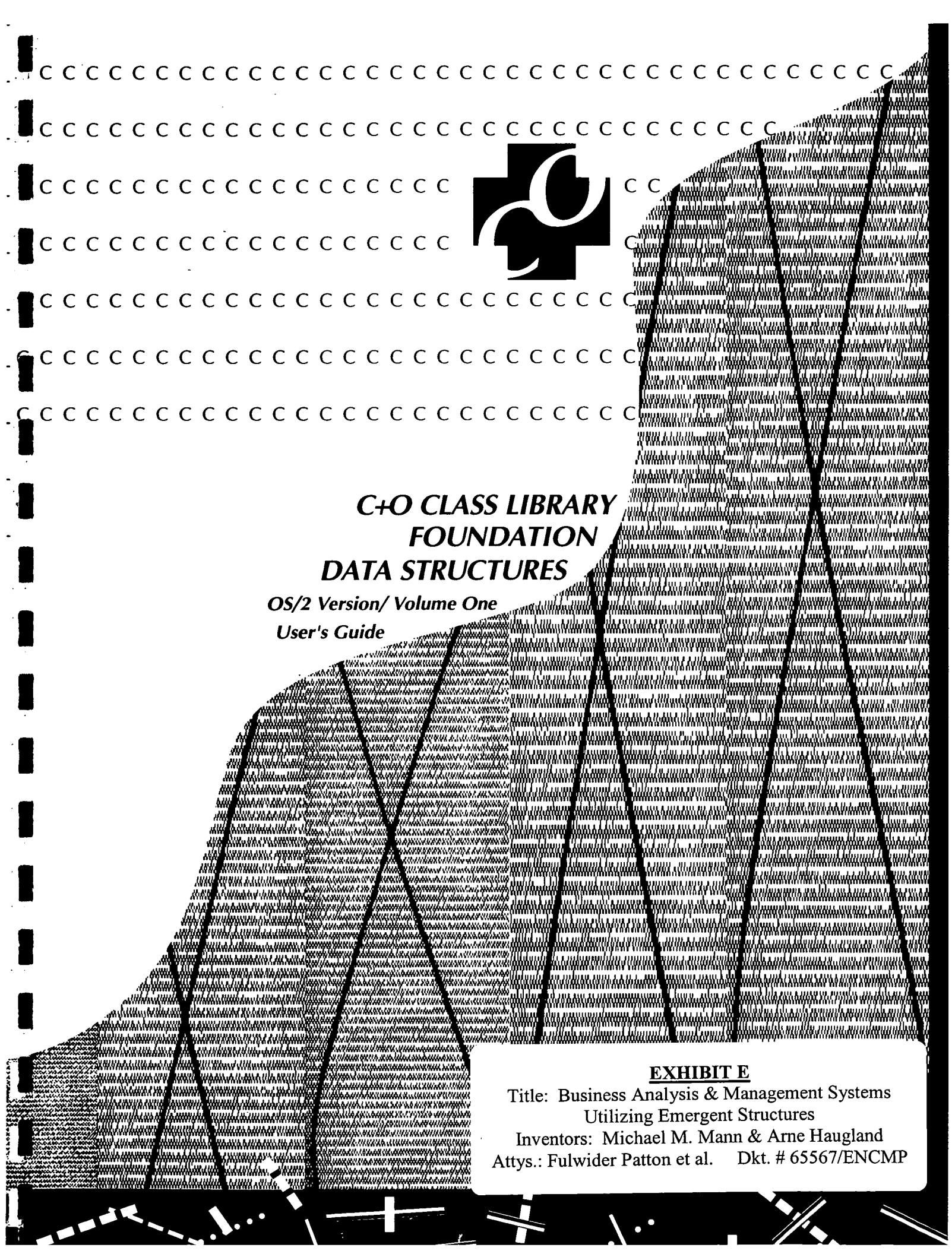


C+O CLASS LIBRARY FOUNDATION DATA STRUCTURES

OS/2 Version/ Volume One

User's Guide

EXHIBIT E

Title: Business Analysis & Management Systems Utilizing Emergent Structures

Inventors: Michael M. Mann & Arne Haugland

Attys.: Fulwider Patton et al. Dkt. # 65567/ENCMP

C+O Class Library
User's Guide
Volume 1: Foundation Data Structures

Revision 1.2 July, 1988

For the OS/2 Operating System

Objective Systems

Objective Systems
2443 Fillmore Street
Suite 249
San Francisco, California 94115

Tel: 415/ 929-0964
Fax: 415/ 929-8015

This manual was produced in its entirety with the Objective Systems Object-Oriented CASE documentation tool and Microsoft Word. Output was to a PostScript printer.

Table of Contents

Table of Contents (cont.)

Introduction

Congratulations on your purchase of C+O Class Libraries, Volume 1: Foundation Data Structures! C+O is a new kind of product called a Class Library. Class libraries are different from other kinds of libraries because they can model general purpose data structures, not just DOS interfaces or user interfaces. Secondly, class libraries are new because they employ object-oriented programming techniques. What makes this so important is that it establishes new levels of reusability.

In creating C+O Class Libraries, we have always kept the following things in mind:

1. Programmers are the toughest customers around -- especially when it comes to using someone else's source code.

2. The documentation is at least as important as the code.

3. Debugging aids are an important mechanism for understanding and learning to use a library.

4. Examples, examples, examples.

5. A picture is worth many pages of text, many hours of debugging and many support calls.

We think you will be pleasantly surprised how much we took these principles to heart.

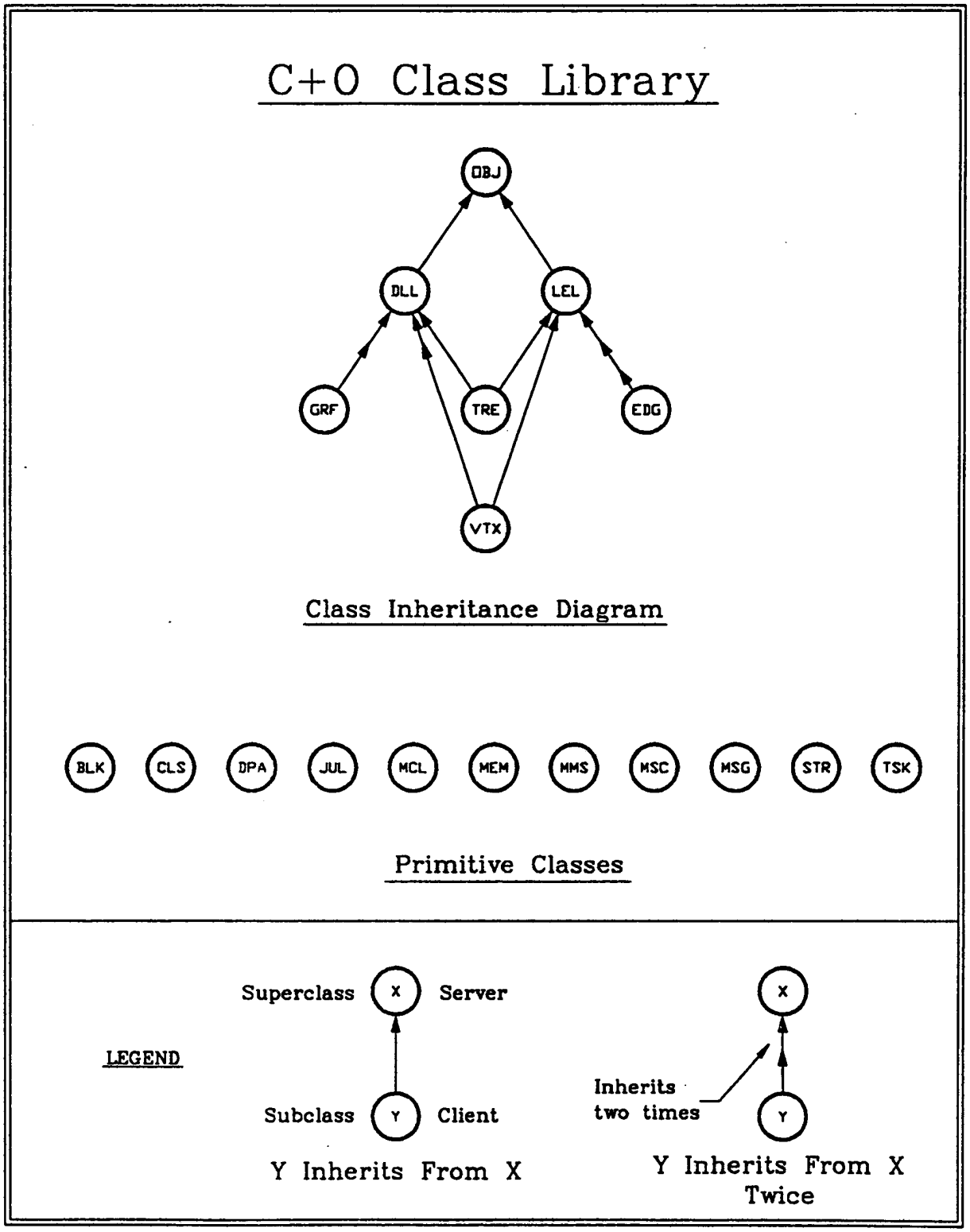
At Objective Systems, we are firmly committed to the concept of reusable code. C+O was originally developed for our own use in developing a variety of commercial products. We believe that object-oriented techniques are the solution to the reusability problem.

At Objective Systems, we are firmly committed to standards. C+O is fully compliant with ANSI C conventions. C+O is named using SAA naming conventions. As standards for object-oriented programming emerge, we will adapt C+O class libraries to those standards as well.

Finally, C+O is a living product. It will be continually enhanced and upgraded. We value your suggestions and comments and will strive to incorporate them in future releases. Over time, we will porting C+O to other hardware and software platforms as well as coming out with new libraries. Please let us know what *you* would like to see.

Should you have any problems or questions regarding C+O class libraries, please give us a call. We are available from 9am to 7pm West coast time.

Class Inheritance Diagram



Quick Reference Guide to Classes

Function Name	Description	Page	Scope	Macro
BlkClear	Clear instance	Blk - 2	Public	N
BlkDeInit	Deinitialize instance	Blk - 3	Public	N
BlkExecute	Execute method with parameters	Blk - 4	Public	N
BlkExecuteRetBool	Execute the method, return int	Blk - 5	Public	N
BlkExecuteRetDataPtr	Execute method, return mem pointer	Blk - 6	Public	N
BlkExecuteRetFuncPtr	Execute method, return function ptr	Blk - 7	Public	N
BlkExecuteRetInt	Execute method, return int	Blk - 8	Public	N
BlkHasMethod	Return True if instance has non-NULL method	Blk - 9	Public	N
BlkInit	Initialize instance	Blk - 10	Public	N
BlkPrint	Print contents of instance	Blk - 11	Public	N
BlkPushDataPtr	Save pointer parameter	Blk - 12	Public	N
BlkPushFuncPtr	Save function pointer parameter.	Blk - 13	Public	N
BlkPushLargeInt	Save LargeInt parameter	Blk - 14	Public	N
BlkPushMediumInt	Save MediumInt parameter	Blk - 15	Public	N
BlkSetMethod	Set the function to call	Blk - 16	Public	N

ClsCreateMessages	Create the Msg instances	N/A	Undoc	N
ClsCreateObject	Create a new Object instance	Cls - 2	Public	N
ClsCreateSupers	Create the superclass instances	N/A	Undoc	N
ClsDeInit	Deinitialize the instance	Cls - 3	Public	N
ClsDestroy	Deallocate the instance	Cls - 4	Public	N
ClsDestroyMessages	Destroy the messages	N/A	Undoc	N
ClsDestroyObject	Deallocate the object instance	Cls - 5	Public	N
ClsDestroySuperClasses	Destroy the superclasses	N/A	Undoc	N
ClsFindMsg	Find message given selector name	Cls - 7	Public	N
ClsFindSelectorIndex	Find index of selector given name	Cls - 8	Public	N
ClsFindSuperClass	Find superclass given name	Cls - 10	Public	N
ClsGetMessageCount	Get number of messages	Cls - 11	Public	N
ClsGetMethodAndOffset	Return method and sub/super offset	Cls - 12	Public	N
ClsGetName	Return name of the instance	Cls - 14	Public	N
ClsGetNthMsg	Get pointer to nth Message	Cls - 15	Public	N
ClsGetNthSuperClass	Get pointer to Nth superclass	Cls - 16	Public	N
ClsGetObjectSize	Return size of an object instance	Cls - 17	Public	N
ClsGetOffsetForMsg	Return sub/super offset of object	Cls - 18	Public	N
ClsGetOffsetOfNthSuper	Return offset of nth superobject	Cls - 20	Public	N
ClsGetRootSubClass	Get the root subclass	Cls - 22	Public	N
ClsGetRootSubObjectOffset	Return offset of root subobject	Cls - 23	Public	N
ClsGetRootSubObjectSize	Return size of the root subobject	Cls - 24	Public	N
ClsGetSize	Return size of the instance	Cls - 25	Public	N
ClsGetSubClass	Return subclass	Cls - 26	Public	N
ClsGetSubObjectOffset	Return offset of a subobject	Cls - 27	Public	N

Quick Reference Guide to Classes

Function Name	Description	Page	Scope	Macro
ClsGetSuperClassCount	Return number of superclasses	Cls - 28	Public	N
ClsGetSuperClassIndex	Return superclass index	Cls - 29	Public	N
ClsInit	Initialize the instance	Cls - 30	Public	N
ClsIsRoot	Is instance the root subclass?	Cls - 31	Public	N
ClsOffsetSupers	Offset superclasses from subclass.	N/A	Undoc	N
ClsPrint	Print the instance	Cls - 32	Public	N
ClsSendDestroy	Send destroy message to instance	Cls - 33	Public	N
ClsSendObjectMessage	Send object a message	Cls - 34	Public	N
ClsSendObjectMessageReturnInt	Send object message, returns MediumInt	Cls - 36	Public	N
ClsSendObjectMessageReturnPtr	Send object message, returns Void *	Cls - 38	Public	N
ClsSuperClassOf	Initialize sub/super relation	N/A	Undoc	N

DllAppend	Append element to list	Dll - 2	Public	N
DllAppendLast	Make element last	Dll - 4	Public	Y
DllAsObj	Return list as object	Dll - 6	Private	Y
DllClear	Clear the list	Dll - 7	Public	N
DllCut	Cut one element from list	Dll - 8	Public	N
DllCutChildren	Cut all elements from list	Dll - 10	Public	N
DllCutRange	Cut element(s) from list	Dll - 11	Public	N
DllDeInit	Deinitialize list object	Dll - 13	Public	N
DllDestroy	Deinitialize list object and free space	Dll - 14	Public	N
DllGetClient	Return client of list	Dll - 15	Public	Y
DllGetFirst	Return first element	Dll - 16	Private	Y
DllGetLast	Return last element	Dll - 17	Private	Y
DllGetNth	Return Nth element	Dll - 18	Private	N
DllGetSize	Get size of list	Dll - 19	Public	N
DllInit	Initialize list object	Dll - 20	Public	N
DllInsert	Insert element in list	Dll - 21	Public	N
DllInsertFirst	Make element first	Dll - 23	Public	Y
DllIsEmpty	Return True if list empty	Dll - 25	Public	Y
DllLelClientCount	Visit function: count elements conditionally	Dll - 26	Public	N
DllLelClientFind	Visit search function: all elements	Dll - 28	Public	N
DllLelClientFirst	Return client of first element	Dll - 30	Public	Y
DllLelClientGetNth	Return Nth client	Dll - 31	Public	N
DllLelClientLast	Return client of last element	Dll - 33	Public	Y
DllLelClientVisitBwd	Visit function: all elements	Dll - 34	Public	N
DllLelClientVisitFwd	Visit function: all elements	Dll - 36	Public	N
DllNotifyCutRange	Cut elements	Dll - 38	Friend	N
DllNotifyPasteRange	Paste elements	Dll - 39	Friend	N
DllPasteRangeAfter	Paste element(s) to list	Dll - 40	Public	N
DllPasteRangeBefore	Paste element(s) in list	Dll - 42	Public	N
DllPasteRangeFirst	Paste element(s) to be first in list	Dll - 44	Public	Y
DllPasteRangeLast	Paste element(s) to end of list	Dll - 46	Public	Y
DllSendDestroy	Send message for list destruction	Dll - 48	Public	N

Quick Reference Guide to Classes

Function Name	Description	Page	Scope	Macro
DpaAppend	Append an element	Dpa - 2	Public	N
DpaClear	Clear dynamic array	Dpa - 4	Public	N
DpaCount	Visit function: count True returns	Dpa - 5	Public	N
DpaCountRange	Visit function: range with return checking	Dpa - 7	Public	N
DpaDeInit	Deinitialize the dynamic array object	Dpa - 9	Public	N
DpaDelete	Delete element(s)	Dpa - 10	Public	N
DpaDestroy	Deinitialize array object and free space	Dpa - 12	Public	N
DpaExpand	Paste Null element(s) into array	Dpa - 13	Public	N
DpaFind	Find index returning True	Dpa - 15	Public	N
DpaFindPtrBwd	Find index with matching pointer	Dpa - 17	Public	N
DpaFindPtrFwd	Find index with matching pointer	Dpa - 19	Public	N
DpaFindRangeBwd	Find index returning True for range	Dpa - 21	Public	N
DpaFindRangeFwd	Find index returning True for range	Dpa - 23	Public	N
DpaGetLast	Return last element in array	Dpa - 27	Public	N
DpaGetNth	Return Nth array element	Dpa - 25	Public	N
DpaGetSize	Return number of elements	Dpa - 29	Public	N
DpaInit	Initialize the edge object	Dpa - 31	Public	N
DpaNewArray	Create a new array	N/A	Undoc	N
DpaResize	Resize the array	N/A	Undoc	N
DpaSetNth	Set Nth element of array	Dpa - 33	Public	N
DpaSetRegionNull	Make region of elements Null	Dpa - 35	Public	N
DpaSetSize	Set array size to N elements	Dpa - 37	Public	N
DpaShiftDown	Shift down N elements in array	Dpa - 38	Public	N
DpaShiftUp	Shift up N elements in array	Dpa - 40	Public	N
DpaVisit	Visit function: all elements	Dpa - 42	Public	N
DpaVisitClient	Visit function: all elements	Dpa - 44	Public	N
DpaVisitRange	Visit function: range of elements	Dpa - 46	Public	N
DpaVisitRegion	Visit function: region of elements	Dpa - 48	Public	N
DpaVisitSelfAndSuccessors	Visit function: client and successors	Dpa - 50	Public	N

EdgAsGrfLel	Return graph list element for edge	Edg - 2	Friend	Y
EdgAsInLel	Return incoming edge list element	Edg - 3	Friend	Y
EdgAsObj	Return edge as object	Edg - 4	Private	N
EdgAsOutLel	Return outgoing edge list element	Edg - 5	Friend	Y
EdgClear	Clear the edge	Edg - 6	Public	N
EdgCompareInVtx	Compare incoming vertex	Edg - 7	Public	N
EdgConnectToGrf	Connect edge to graph	Edg - 9	Public	Y
EdgConnectToVertices	Connect edge to vertices	Edg - 11	Public	N
EdgDeInit	Deinitialize the edge object	Edg - 13	Public	N
EdgDestroy	Deinitialize edge object and free space	Edg - 14	Public	N
EdgDisconnectFromGrf	Disconnect edge from graph	Edg - 15	Public	Y
EdgDisconnectFromVertices	Disconnect edge from vertices	Edg - 17	Public	N

Quick Reference Guide to Classes

Function Name	Description	Page	Scope	Macro
EdgGetClient	Return client of edge	Edg - 19	Public	Y
EdgGetGrf	Return graph	Edg - 20	Friend	Y
EdgGetInVtx	Return incoming vertex	Edg - 21	Friend	Y
EdgGetNextIn	Return next incoming edge	Edg - 23	Friend	Y
EdgGetNextOut	Return next outgoing edge	Edg - 25	Friend	Y
EdgGetOutVtx	Return outgoing vertex	Edg - 27	Friend	Y
EdgGetVertices	Return vertices to edge	Edg - 29	Public	Y
EdgHasVertices	Does edge have any vertices	Edg - 31	Public	N
EdgInGrf	Is edge in graph	Edg - 32	Public	Y
EdgInit	Initialize the edge object	Edg - 33	Public	N
EdgSendDestroy	Send message for edge destruction	Edg - 34	Public	Y
EdgUpdateInVtx	Replace incoming vertex	Edg - 35	Public	N
EdgUpdateOutVtx	Replace outgoing vertex	Edg - 37	Public	N
GrfAnyCycles	Check graph for cycles	Grf - 2	Public	Y
GrfAsEdgDll	Return List of edges	Grf - 4	Friend	N
GrfAsObj	Return graph as object	Grf - 5	Private	N
GrfAsVtxDll	Return List of vertices	Grf - 6	Friend	N
GrfBasicTopologicalSort	Do topological sort of graph	N/A	Undoc	N
GrfClear	Clear the graph	Grf - 7	Public	N
GrfCountEdg	Count edges of graph	Grf - 8	Public	Y
GrfCountVtx	Count vertices of graph	Grf - 10	Public	Y
GrfDeInit	Deinitialize Graph object	Grf - 12	Public	N
GrfDestroy	Deinitialize Graph object and free space	Grf - 13	Public	N
GrfDoTopologicalSort	Do topological sort of graph	Grf - 14	Public	Y
GrfFindEdgClient	Visit search function: edges	Grf - 16	Public	N
GrfFindVtxClient	Visit search function: vertices	Grf - 18	Public	N
GrfGetClient	Return client of graph	Grf - 20	Public	N
GrfInit	Initialize Graph object	Grf - 21	Public	N
GrfSendDestroy	Send message for graph destruction	Grf - 22	Public	N
GrfVisitEdgClient	Visit function: edges	Grf - 23	Public	N
GrfVisitVtxClient	Visit function: vertices	Grf - 25	Public	N
GrfVisitVtxClientInTopOrderBwd	Visit function: backward topological order	Grf - 27	Public	N
GrfVisitVtxClientInTopOrderFwd	Visit function: forward topological order	Grf - 29	Public	N
JulAddDays	Add/subtract days to date	Jul - 2	Public	Y
JulAddDaysL	Add/subtract days to date (long)	Jul - 4	Public	Y
JulAddMonths	Add/subtract months to date	Jul - 6	Public	N
JulAddQuarters	Add/subtract quarters to date	Jul - 8	Public	N
JulAddYears	Add/subtract years to date	Jul - 10	Public	N
JulCalendarToJulian	Day, month, year to julian day	Jul - 12	Public	N

Quick Reference Guide to Classes

Function Name	Description	Page	Scope	Macro
JulCopy	Copy julian day	Jul - 14	Public	Y
JulDateStrToJulian	Date string to julian day	Jul - 16	Public	N
JulDayOfWeek	Day number in week	Jul - 18	Public	Y
JulDayOfYear	Day number in year	Jul - 20	Public	N
JulDaysInMonth	Days in month	Jul - 22	Public	N
JulDaysInQuarter	Days in quarter	Jul - 24	Public	N
JulDaysInYear	Days in year	Jul - 26	Public	N
JulDiff	Days between two dates	Jul - 28	Public	Y
JulDiffL	Days between two dates (long)	Jul - 30	Public	Y
JulGetSystemJulianDay	System date as julian day	Jul - 32	Public	N
JulInit	Set to beginning of calendar January 1, 1583	Jul - 34	Public	Y
JulIsLeapYear	Is date in leap year	Jul - 36	Public	N
JulIsMaxValue	Is date maximum julian value	Jul - 38	Public	Y
JulMax	The maximum of two julian dates	Jul - 40	Public	Y
JulMin	The minimum of two julian dates	Jul - 42	Public	Y
JulMonthDayDiff	Days between date and a day/month	Jul - 44	Public	N
JulMonthString	Fill string with month and year	Jul - 46	Public	N
JulQuarterString	Fill string with quarter and year	Jul - 48	Public	N
JulSameDayMonth	Two dates same day and month	Jul - 50	Public	N
JulSetMaxDate	Set date to maximum value	Jul - 52	Public	N
JulToCalendar	Julian day to day, month, year	Jul - 54	Public	N
JulToDateStr	Fill date string of specified format	Jul - 56	Public	N
JulValidateDate	Validate date passed as string	Jul - 58	Public	N
JulWeekString	Fill string with day and month	Jul - 60	Public	N
JulYearString	Fill string with year	Jul - 62	Public	N

LelAsObj	Return element as object	Lel - 2	Private	Y
LelClientCount	Return count for client and successors	Lel - 3	Public	N
LelClientDll	Return client of list	Lel - 5	Public	Y
LelClientFindRange	Visit search function: range	Lel - 6	Public	N
LelClientNext	Return client of next element	Lel - 8	Public	Y
LelClientPrev	Return client of previous element	Lel - 9	Public	Y
LelClientVisitBwd	Visit function: client and predecessors	Lel - 10	Public	N
LelClientVisitFwd	Visit function: client and successors	Lel - 12	Public	N
LelClientVisitPredecessors	Visit function: predecessors	Lel - 14	Public	N
LelClientVisitRange	Visit function: range	Lel - 16	Public	N
LelClientVisitSuccessors	Visit function: successors	Lel - 18	Public	N
LelCountRange	Count elements	Lel - 20	Public	N
LelCut	Cut element from list	Lel - 22	Public	N
LelCutRange	Cut element(s) from list	Lel - 24	Public	N
LelCutRangeFromList	Cut element(s) from list	N/A	Undoc	N
LelDeInit	Deinitialize list element object	Lel - 26	Public	N

Quick Reference Guide to Classes

Function Name	Description	Page	Scope	Macro
LelDestroy	Deinitialize list element object and free space	Lel - 27	Public	N
LelGetClient	Return client	Lel - 28	Public	Y
LelGetDll	Return list object	Lel - 29	Friend	Y
LelGetNext	Return next element	Lel - 30	Friend	Y
LelGetNthSuccessor	Return Nth element	Lel - 31	Friend	N
LelGetPrev	Return previous element	Lel - 32	Friend	Y
LelInit	Is element in list	Lel - 33	Public	N
LelPasteRangeAfter	Append elements(s) to list	Lel - 34	Public	N
LelPasteRangeBefore	Insert element(s) to list	Lel - 36	Public	N
LelPasteRangeToList	Insert element(s) to list	N/A	Undoc	N
LelSendDestroy	Send message for list element destruction	Lel - 38	Public	N
LelTest	Test for valid list element	Lel - 39	Public	N
LelVisitRange	Visit function for range	Lel - 40	Private	N

MclCreateClass	Create a class instance	Mcl - 2	Private	N
MclDestroyClass	Destroy a class instance	Mcl - 3	Friend	N
MclFindSelector	Find a selector index given its name	Mcl - 5	Public	N
MclFindSuperClass	Find the superclass index given its name	Mcl - 6	Public	N
MclGetClassName	Get the class name	Mcl - 7	Public	N
MclGetClassSize	Return the size of a class instance	Mcl - 8	Public	N
MclGetMessageCount	Return the number of messages	Mcl - 9	Public	N
MclGetNthMms	Return the Nth MetaMessage	Mcl - 10	Public	N
MclGetNthOffset	Return the Nth SuperClass offset	Mcl - 11	Public	N
MclGetNthSuper	Return the MetaClass	Mcl - 12	Public	N
MclGetSuperClassCount	Return the number of superclasses	Mcl - 13	Public	N
MclPrint	Print the MetaClass instance	Mcl - 14	Public	N
MclSendCreateClass	Return a new instance of the MetaClass	Mcl - 15	Public	N
MclSendDestroyClass	Deallocate an instance of a Class	Mcl - 16	Public	N
MclValidate	Validate that the instance is valid	Mcl - 17	Public	N
MclValidateMessages	Validate the MetaMessages	N/A	Undoc	N
MclValidateSuperClasses	Validate the MetaSuperClasses	N/A	Undoc	N

MemClear	Clears array elements	Mem - 2	Public	N
MemCopy	Copy array elements within the array	Mem - 3	Public	N
MemCutNSet	Delete elements, shift remaining	Mem - 5	Public	N
MemDestroy	Deallocate memory	Mem - 6	Public	N
MemDuplicate	Copy array into another	Mem - 7	Public	N
MemNew	Allocate memory	Mem - 8	Public	N
MemPasteNSet	Expand array, shift remaining	Mem - 9	Public	N
MemSetChr	Set elements to character	Mem - 10	Public	N

Quick Reference Guide to Classes

Function Name	Description	Page	Scope	Macro
MmsGetMethod	Get the method	Mms - 2	Public	N
MmsGetSelector	Get the name of selector	Mms - 6	Public	N
MmsGetSuper	Get name of superclass	Mms - 4	Public	N
MmsPrint	Display the instance	Mms - 7	Public	N
MmsValidate	Validate the instance	Mms - 8	Public	N
MscGetMetaClass	Get the MetaClass	Msc - 3	Public	N
MscGetName	Get the name of superclass	Msc - 2	Public	N
MscGetOffset	Get offset of superclass	Msc - 4	Public	N
MscPrint	Display the contents of the MetaSuperClass	Msc - 5	Public	N
MscValidate	Validate that the instance is valid	Msc - 6	Public	N
MsgDeInit	Deinit instance	Msg - 2	Public	N
MsgGetMethod	Return method	Msg - 4	Public	N
MsgGetOffset	Return the object offset	Msg - 3	Public	N
MsgGetSelector	Return selector	Msg - 5	Public	N
MsgInit	Initialize instanc	Msg - 6	Public	N
MsgPrint	Print the instance	Msg - 7	Public	N
MsgSend	Send message to object	Msg - 8	Public	N
MsgSendReturnInt	Send message to object, return int	Msg - 9	Public	N
MsgSendReturnPtr	Send message to object, return pointer	Msg - 10	Public	N
MsgSetSuperOffsetAndMethod	Override the method	N/A	Undoc	N
ObjDeInit	Deinitialize instance	Obj - 2	Public	N
ObjDestroy	Deinitialize and deallocate object	Obj - 3	Public	N
ObjGetClient	Return a client	Obj - 4	Public	N
ObjGetClientOrNull	Return client if non-NULL	Obj - 6	Public	N
ObjGetCls	Return class	Obj - 8	Public	N
ObjGetClsName	Return class name	Obj - 9	Public	N
ObjGetImmediateClient	Return immediate client	Obj - 10	Public	N
ObjGetMethodAndOffset	Return method and client offset	Obj - 11	Public	N
ObjGetNthSuperObject	Return nth superobject	Obj - 13	Public	N
ObjGetRootClient	Get root client	Obj - 14	Public	N
ObjGetRootClientSize	Get root object size	Obj - 15	Public	N
ObjGetSize	Get object size	Obj - 16	Public	N
ObjGetSubObjectOffset	Return offset to immediate subobject	Obj - 17	Public	N
ObjInit	Initialize object	Obj - 18	Public	N
ObjIsRoot	Return Treu if is root subobject	Obj - 19	Public	N
ObjRespondsToSelector	Does object understand message?	Obj - 20	Public	N

Quick Reference Guide to Classes

Function Name	Description	Page	Scope	Macro
ObjSendMessage	Send message to object	Obj - 21	Public	N
ObjSendMessageReturnInt	Send message to object, return Int	Obj - 22	Public	N
ObjSendMessageReturnPtr	Send message to object, return pointer	Obj - 23	Public	N

StrBasicExtract	Extract a string	N/A	Undoc	N
StrExtract	Extract string as specified	Str - 2	Public	N
StrFromDate	Fill string with a date	Str - 4	Public	N
StrFromMediumInt	Integer to string	Str - 6	Public	N
StrInit	Init string to zero	Str - 8	Public	Y
StrReplaceSubStr	Replace sub-string in string	Str - 9	Public	N
StrSet	Copy string to another	Str - 11	Public	Y
StrSqueeze	Removes any character from string	Str - 13	Public	N
StrToDate	Parse a date string for year, month, day	Str - 15	Public	N
StrToLower	Change case of string to lower	Str - 17	Public	Y
StrToMediumInt	String to integer	Str - 19	Public	N
StrToUpper	Change case of string to upper	Str - 21	Public	Y

TreAsDll	Return node as list	Tre - 2	Private	Y
TreAsLel	Return node as list element	Tre - 3	Private	Y
TreAsObj	Return node as object	Tre - 4	Private	Y
TreClear	Clear the tree	Tre - 5	Public	N
TreClient	Return client of node	Tre - 6	Public	Y
TreClientFindChild	Visit search function: children	Tre - 7	Public	N
TreClientFirstChild	Return first child node as client	Tre - 9	Public	Y
TreClientLastChild	Return last child node as client	Tre - 11	Public	Y
TreClientLastLeaf	Return last leaf node as client	Tre - 13	Public	Y
TreClientNext	Return next node as client	Tre - 15	Public	Y
TreClientNextPreOrder	Return next PreOrder client node	Tre - 17	Public	Y
TreClientNextUncle	Return next uncle as client	Tre - 19	Public	Y
TreClientParent	Return parent node	Tre - 21	Public	Y
TreClientPrev	Return prev client	Tre - 23	Public	Y
TreClientPrevPreOrder	Return previous client PreOrderly	Tre - 25	Public	Y
TreClientVisitBranchInOrder	Visit function: branch in-order	Tre - 27	Public	N
TreClientVisitChildren	Visit function: all children	Tre - 29	Public	N
TreClientVisitChildrenBwd	Visit function: all children	Tre - 31	Public	N
TreClientVisitDescBranchInOrder	Visit function: descendents	Tre - 33	Public	N
TreClientVisitDescInOrder	Visit function: descendents	Tre - 35	Public	N
TreClientVisitDescInOrderBwd	Visit function: descendents	Tre - 37	Public	N
TreClientVisitDescLeaves	Visit function: descendents	Tre - 39	Private	N
TreClientVisitDescPreOrder	Visit function: descendents	Tre - 41	Public	N
TreClientVisitInOrder	Visit function: in-order	Tre - 43	Public	N
TreClientVisitInOrderBwd	Visit function: in-order	Tre - 45	Public	N

Quick Reference Guide to Classes

Function Name	Description	Page	Scope	Macro
TreClientVisitLeaves	Visit function: leaves	Tre - 47	Public	N
TreClientVisitParents	Visit function: nearest parents first	Tre - 49	Public	N
TreClientVisitPreOrder	Visit function: pre-order	Tre - 51	Public	N
TreClientVisitRange	Visit function: range	Tre - 53	Public	N
TreClientVisitSuccPreOrder	Visit function: all successors	Tre - 55	Public	N
TreClientVisitSuccessors	Visit function: successors	Tre - 57	Public	N
TreCutChildren	Cut children from tree	Tre - 59	Public	Y
TreCutRange	Cut node(s) from tree	Tre - 61	Public	Y
TreDeInit	Deinitialize Tree object	Tre - 63	Public	N
TreDestroy	Deinitialize Tree object and free space	Tre - 64	Public	N
TreDestroyChildren	Destroy any children of a tre	Tre - 65	Public	N
TreFirstChild	Return first child	Tre - 66	Private	Y
TreHasChildren	Does node have any children	Tre - 68	Public	Y
TreHasSiblings	Does node have any siblings	Tre - 69	Public	Y
TreInit	Initialize tree object	Tre - 70	Public	N
TreIsChild	Does the node have a parent	Tre - 71	Public	Y
TreIsDirectAncestor	Is node a direct ancester	Tre - 72	Public	N
TreIsRoot	Does the node have no parent	Tre - 73	Public	Y
TreLastChild	Return last child	Tre - 74	Private	Y
TreLastLeaf	Return last leaf	Tre - 76	Private	N
TreNext	Return next node	Tre - 78	Private	Y
TreNextPreOrder	Return next node PreOrderly	Tre - 80	Private	N
TreNextUncle	Return next uncle	Tre - 82	Private	N
TreParent	Return parent node	Tre - 84	Private	Y
TrePasteRangeAfterSibling	Paste range of siblings	Tre - 86	Public	Y
TrePasteRangeBeforeSibling	Paste range of siblings	Tre - 88	Public	Y
TrePasteRangeFirstChild	Paste children	Tre - 90	Public	Y
TrePasteRangeLastChild	Paste children	Tre - 92	Public	Y
TrePrev	Return previous node	Tre - 94	Private	Y
TrePrevPreOrder	Return previous node PreOrderly	Tre - 96	Private	N
TreSendDestroy	Send message for tree destruction	Tre - 98	Public	N
TreVisitBranchInOrder	Visit function: branch in-order	Tre - 99	Private	N
TreVisitChildren	Visit function: children	Tre - 101	Private	N
TreVisitChildrenBwd	Visit function: children	Tre - 103	Private	N
TreVisitDescBranchInOrder	Visit function: descendents	Tre - 105	Private	N
TreVisitDescInOrder	Visit function: descendents	Tre - 107	Private	N
TreVisitDescInOrderBwd	Visit function: descendents	Tre - 109	Private	N
TreVisitDescPreOrder	Visit function: descendents	Tre - 111	Private	N
TreVisitInOrder	Visit function: in-order	Tre - 113	Private	N
TreVisitInOrderBwd	Visit function: in-order	Tre - 115	Private	N
TreVisitLeaves	Visit function: leaves	Tre - 117	Private	N
TreVisitParents	Visit function: nearest parents first	Tre - 119	Private	N
TreVisitPreOrder	Visit function: pre-order	Tre - 121	Private	N
TreVisitRange	Visit function: range	Tre - 123	Private	N
TreVisitSuccPreOrder	Visit function: all successors	Tre - 125	Private	N
TreVisitSuccessors	Visit function: successors	Tre - 127	Private	N

Quick Reference Guide to Classes

Function Name	Description	Page	Scope	Macro
TskBasicAssert	Foundation function	N/A	Undoc	N
TskCondition	Raise exception conditionally	Tsk - 2	Public	N
TskDeInit	Deinitialize instance	Tsk - 4	Public	N
TskDefaultInit	Initialize instance with defaults	Tsk - 5	Public	N
TskExit	Exit program with code	Tsk - 6	Public	N
TskExitWithMsg	Exit program and print message	Tsk - 7	Public	N
TskGetArgc	Get main() argument count	Tsk - 8	Public	N
TskGetArgv	Get main() argument vector	Tsk - 9	Public	N
TskGetExceptionCondition	Return exception condition	Tsk - 10	Public	N
TskGetExceptionFileName	Return filename of exception	Tsk - 11	Public	N
TskGetExceptionLineNo	Return exception line number	Tsk - 12	Public	N
TskGetExceptionType	Return exception type	Tsk - 13	Public	N
TskInit	Initialize instance	Tsk - 15	Public	N
TskIsInitialized	Is the Task initialized	N/A	Undoc	N
TskLogCond	Raise exception conditionally	Tsk - 17	Public	N
TskMainLogCond	Raise exception conditionally	Tsk - 19	Public	N
TskMainPreCond	Raise exception conditionally	Tsk - 21	Public	N
TskMainPtrCond	Raise exception conditionally	Tsk - 23	Public	N
TskMainRaiseException	Raise exception unconditionally	Tsk - 25	Public	N
TskNormalExit	Exit task normally	Tsk - 27	Public	N
TskOnException	Establish exception handler	Tsk - 28	Public	N
TskPopExceptionHandler	Pop exception handler	Tsk - 30	Public	N
TskPreCond	Raise exception conditionally	Tsk - 32	Public	N
TskPrintException	Print description of exception	Tsk - 34	Public	N
TskPropagateException	Pass last exception	Tsk - 36	Public	N
TskPtrCond	Raise exception conditionally	Tsk - 38	Public	N
TskPushExh	Push the exception	N/A	Undoc	N
TskRaiseException	Raise exception unconditionally	Tsk - 40	Public	N

VtxAsGrfLel	Return list element in graph	Vtx - 2	Friend	Y
VtxAsInDll	Return vertex as list of incoming edges	Vtx - 3	Friend	N
VtxAsObj	Return edge as object	Vtx - 4	Private	N
VtxAsOutDll	Return vertex as list of outgoing edges	Vtx - 5	Friend	N
VtxClear	Clear vertex	Vtx - 6	Public	N
VtxConnectToGrf	Connect vertex to graph	Vtx - 7	Public	N
VtxCountIn	Count incoming edges	Vtx - 9	Public	Y
VtxCountOut	Count outgoing edges	Vtx - 11	Public	Y
VtxDeInit	Deinitialize the Vertex object	Vtx - 13	Public	N
VtxDestroy	Deinitialize Vertex object and free space	Vtx - 14	Public	N
VtxDisconnectFromGrf	Disconnect vertex from graph	Vtx - 15	Public	Y
VtxFindOutEdg	Visit search function: outgoing edges	Vtx - 17	Private	N
VtxFindOutEdgClient	Visit search function: outgoing edges	Vtx - 19	Public	N
VtxGetClient	Return client of vertex	Vtx - 21	Public	Y

Function Name	Description	Page	Scope	Macro
VtxGetFirstIn	Return first incoming edge	Vtx - 22	Public	Y
VtxGetFirstOut	Return first outgoing edge	Vtx - 23	Public	Y
VtxGetGrf	Return graph	Vtx - 24	Friend	Y
VtxInGrf	Is vertex in graph	Vtx - 26	Public	Y
VtxInit	Initialize the Vertex object	Vtx - 25	Public	N
VtxSendDestroy	Send message for vertex destruction	Vtx - 27	Public	N
VtxStackSetup	Set up values for topsort	Vtx - 28	Friend	Y
VtxVisitEdge	Visit function: each edge	Vtx - 29	Friend	N
VtxVisitEdgeClient	Visit function: each edge	Vtx - 31	Public	N
VtxVisitInEdge	Visit function: incoming edge	Vtx - 33	Friend	N
VtxVisitInEdgeClient	Visit function: incoming edge	Vtx - 35	Public	N
VtxVisitOutEdge	Visit function: outgoing edge	Vtx - 37	Friend	N
VtxVisitOutEdgeClient	Visit function: outgoing edge	Vtx - 39	Public	N

Quick Reference Guide to Classes

This page is intentionally left blank

C+O Datatypes

Name	Size	Definition	Usage
Blk	20/38	struct *Block*	*Block* class
Block	20/38	struct *Block*	*Block* class
Bool	1	char	Boolean values: True or False
Call	N/A	(Void)	Overrides function returning value
Char	1	char	Text data
Chr	1	char	Text data
Class	16/26	struct *Class*	*Class* class
ClientPtr	2/4	Void *	Data pointer to unknown type
Cls	16/26	struct *Class*	*Class* class
Const	N/A	const	Identifies function parameters which are not modified
DateFormat	2	enum DateFormat	Describe a display format for dates (see *JulianTime* and *String*)
Dll	6/12	struct *List*	*List* class
Dpa	8/10	struct *DynamicArray*	*DynamicArray* class
DynamicArray	8/10	struct *DynamicArray*	*DynamicArray* class
Edg	26/52	struct *Edge*	*Edge* class
Edge	26/52	struct *Edge*	*Edge* class
ExceptionType	2	enum ExceptionType	Type of exception generated (see *Task*)
ExcFilter	2/4	Bool (*)(PTSK)	Function pointer to exception filters (see *Task*)
Ext	2	enum ExceptionType	Type of exception generated
False	N/A	((Bool)0)	Boolean False
Flags8	1	unsigned char	Bit flags (8)
Flags16	2	unsigned short	Bit flags (16)
Flags32	4	unsigned char	Bit flags (32)
GenericPtr	2/4	Void *	Data pointer to unknown type
Graph	30/48	struct *Graph*	*Graph* class
Grf	30/48	struct *Graph*	*Graph* class
IntAddress	2/4	int / long	Integer capable of holding a data address
Jul	4	struct *JulianTime*	*JulianTime* class
JulianTime	4	struct *JulianTime*	*JulianTime* class
LARGE_DATA_PTRS	N/A	N/A	Defined if 4 byte data pointers are the default
LARGE_FUNC_PTRS	N/A	N/A	Defined if 4 byte function pointers are the default
LargeInt	4	long	Integers in the range [-2147483647:2147483647]
Lel	8/16	struct *ListElement*	*ListElement* class
LINE_NO	N/A	__LINE__	Line number in file being compiled
List	6/12	struct *List*	*List* class
ListElement	8/16	struct *ListElement*	*ListElement* class
Mcl	20/30	struct *MetaClass*	*MetaClass* class
MediumInt	2	int	Integers in range [-32767:32767]

C+O Datatypes

Name	Size	Definition	Usage
Mem	1	char	*Memory* class
Memory	1	char	*Memory* class
Message	12/22	struct *Message*	*Message* class
MetaClass	20/20	struct *MetaClass*	*MetaClass* class
MetaMessage	8/14	struct *MetaMessage*	*MetaMessage* class
MetaSuperClass	12/22	struct *MetaSuperClass*	*MetaSuperClass* class
Method	2/4	Void (*)(POBJ, ...)	Function pointer (see *Block*)
MethodRetBool	2/4	Bool (*)(POBJ, ...)	Function (returning Bool) pointer (see *Block*)
MethodRetInt	2/4	MediumInt (*)(POBJ, ...)	Function (returning MediumInt) pointer (see *Block*)
MethodRetDataPtr	2/4	PMEM (*)(POBJ, ...)	Function (returning data pointer) pointer (see *Block*)
MethodRetFuncPtr	2/4	PMTH (*)(POBJ, ...)	Function (returning function pointer) pointer (see *Block*)
MethodRetPtr	2/4	Void * (*)(POBJ, ...)	Function (returning pointer) pointer (see *Block*)
Mms	8/14	struct *MetaMessage*	*MetaMessage* class
MODULE_NAME	N/A	__FILE__	Name of file being compiled
Msc	12/22	struct *MetaSuperClass*	*MetaSuperClass* class
Msg	12/22	struct *Message*	*Message* class
NULL	2/4	0	Used to assign or return null pointers
Obj	2/4	struct *Object*	*Object* class
Object	2/4	struct *Object*	*Object* class
PBLK	2/4	struct *Block* *	*Block* pointer - does not require structure definition
PCIO	2/4	struct ConsoleInputOutput *	ConsoleInputOutput pointer - does not require structure definition
PCLS	2/4	struct *Class* *	*Class* pointer - does not require structure definition
PDLL	2/4	struct *List* *	*List* pointer - does not require structure definition
PDPA	2/4	struct *DynamicArray* *	*DynamicArray* pointer - does not require structure definition
PEDG	2/4	struct *Edge* *	*Edge* pointer - does not require structure definition
PGRF	2/4	struct *Graph* *	*Graph* pointer - does not require structure definition
PJUL	2/4	struct *JulianTime* *	*JulianTime* pointer - does not require structure definition
PLEL	2/4	struct *ListElement* *	*ListElement* pointer - does not require structure definition
PMCL	2/4	struct *MetaClass* *	*MetaClass* pointer - does not require structure definition
PMEM	2/4	Char *	Mem pointer - does not require structure definition
PMMS	2/4	struct *MetaMessage* *	*MetaMessage* pointer - does not require structure definition
PMSC	2/4	struct *MetaSuperClass* *	*MetaSuperClass* pointer - does not require structure definition
PMSG	2/4	struct *Message* *	*Message* pointer - does not require structure definition
PMTH	2/4	Void (*)(POBJ, ...)	Function pointer (see *Block*)
POBJ	2/4	struct *Object* *	*Object* pointer - does not require structure definition. Also synonymous with a class pointer of unknown type
PSTR	2/4	char *	Null terminated text strings
PTRE	2/4	struct *Tree* *	*Tree* pointer - does not require structure definition
PTSK	2/4	struct *Task* *	*Task* pointer - does not require structure definition
PVTX	2/4	struct *Vertex* *	*Vertex* pointer - does not require structure definition
Real	8	double	Floating point (no range specified)
Reg1	N/A	register	Prioritized register allocation

Name	Size	Definition	Usage
Reg2	N/A	register	Prioritized register allocation
Reg3	N/A	register	Prioritized register allocation
Reg4	N/A	register	Prioritized register allocation
Reg5	N/A	register	Prioritized register allocation
SmallInt	1	char	Integers in range [-127:+127]
Str	1	char	Text data
String	1	char	Text data
Task	24/32	struct *Task*	*Task* class
Tre	16/32	struct *Tree*	*Tree* class
Tree	16/32	struct *Tree*	*Tree* class
True	N/A	((Bool)1)	Boolean True
Tsk	24/32	struct *Task*	*Task* class
ULargeInt	4	unsigned long	Integers in the range [0:0xFFFFFFFF]
UMediumInt	2	unsigned int	Integers in the range [0:0xFFFF]
USmallInt	1	unsigned char	Integers in range [0:0xFF]
Vertex	22/44	struct *Vertex*	*Vertex* class
Void	N/A	void	Function declarations
Volatile	N/A	volatile	(default) Identifies function parameters which are modified
Vtx	22/44	struct *Vertex*	*Vertex* class

This page is intentionally left blank

Getting Started

This section explains how to get C+O class libraries installed, what the directory structure looks like, source code organization, library organization and most important, organization of the C+O Documentation Set.

Installing C+O class libraries is easy and you will be up in running quickly. However, hold back on the temptation of jumping to the install subheading, you're almost there! Once you have installed C+O, take a few minutes to familiarize yourself with the directory organization of C+O, you will probably want to view certain files as you're reading through the rest of the documentation. Also spend a few minutes thumbing through the manual. Get comfortable with its organization before you begin reading it in depth.

Manual Organization

C+O is divided into two books, the User's Guide and the Reference Manual. This is the User's Guide. The Reference Guide, is a concise description of all the functions supported by C+O.

The diagram at the front of this manual, *Class Inheritance Diagram*, gives a picture of how C+O classes inherit behavior (this will make sense once you've read the rest of the User's Guide).

The table, *C+O Datatypes*, lists all the #defines and typedefs used by C+O. Refer back to this table whenever you encounter a keyword or datatype name you are not familiar with while looking at C+O code or reading this manual.

The table, *Quick Reference: Class Functions*, gives an alphabetical listing of all the functions in this library and a short description. This is handy as a fast way of locating a function.

The User's Guide should be read first. This section you are reading now will help you install C+O and figure out where to look for other information.

Section Two of the User's Guide, *Writing a C+O Program*, gives you a quick start on writing a program using C+O class libraries. You might want to read this section quickly to get an overview, read section Three, then come back to section Two.

Section Three of the User's Guide, *The Object-Oriented Approach to Developing Software*, describes the concepts, methods, and techniques you will need to understand before writing substantial applications using C+O. Once you have read through this, we recommend that you examine the example source code

provided with C+O. The last subsection in this section, *Test-Driving Classes*, will tell you where to find that code and how to compile and link it.

The *Class Summary* is the place to find out what a class is used for, how it is implemented, what functions are available, and what other classes may be related. Spend some time reading through this section, this is where to go when you are looking for a particular class.

Appendix A, *Class Exceptions Reference*, is a complete listing of all the exceptions which can be generated by C+O. It is organized by function name. Within each function it lists the file name and line number where the exception was detected along with a message describing what the exception means. The file name and line number is what will be displayed on your screen when running a C+O program and an exception is detected and trapped by an exception handler. You will refer to this section often as you begin writing your first C+O program.

Once you have made it through the User's Guide, you will be ready to go through the Reference Manual. The Reference Manual consists of the *Class Inheritance Diagram*, the *Quick Reference*, the *C+O Datatypes Table*, and the *Class Reference*. The first three items are the same as described for the User's Guide.

The *Class Reference* is where to go to find out the details of a particular function. Each function occupies a minimum of one page and often two. Where useful, diagrams or examples are provided. This section is arranged alphabetically by function name.

Installing C+O Class Libraries

To install C+O you should have 2.0 Megabytes of hard disk storage available. If you have the source code option, you will need a total of 3.0 Megabytes.

While it is possible to install C+O class libraries on a floppy disk, it is not recommended. We cannot recommend a floppy disk organization that will work for everyone so if you are intent on installing this on a floppy system, you will have to manually copy files over to your working disks. The install procedure provided assumes that all files can be copied onto a single disk.

To install C+O on your hard disk system, change your current drive to A and type INSTALL followed by the fully qualified path you wish to install under. We recommend installing it to the directory COBJECTS.

```
C:> A:
A:> INSTALL C:\COBJECTS
```

Installing C+O on a Hard Disk

C+O will create the necessary directories and copy its files, prompting you to change disks when appropriate. You may abort the install at any time and restart it if need be. C+O is not copy protected and will not put any hidden files on your disk.

For convenience in referring to the install directory, we assume you have installed C+O to C:\COBJECTS. If you installed to another disk or directory, just substitute the name you installed to wherever we mention C:\COBJECTS.

INSTALL creates four subdirectories under C:\COBJECTS. These directories are:

LIB	Microsoft Library files (.LIB extension) containing C+O object code for different memory models and debug and production executables.
INC	C+O include files (.H extension) used for compiling C+O programs.
TST	C source files (.C extension) which can be compiled to test C+O functions and demonstrate how to use C+O classes. Also includes link files (.LNK extension) and batch files (.BAT extension) for compiling and linking.
SRC	C source files (.C extension) which implement the C+O classes. This directory is created only if you have purchased the source code option. Also includes batch files (.BAT extension) for compiling C+O libraries.

Directory structure of C:\COBJECTS

Source Code Organization

The source code provided with C+O consists entirely of C code, namely .C and .H files. C+O source code is divided into classes. Each class has a long name and a three letter abbreviation. The abbreviation, or prefix is used in naming functions, naming variables, and naming the source code files. C+O source code has the following naming convention and associated meanings. The table shown uses a hypothetical class with the prefix Xyz.

xyz.h	This file holds the structure declaration for the class whose abbreviation is Xyz. This file is included if you need the structure definition of Xyz.
xyzmac.h	This file holds the function declarations and macros for the class Xyz. This file is included if you need to perform functions against an object of class Xyz. It automatically includes xyz.h.
xyzcls.h	This file holds any function declarations and macros which are private to the class. It also holds the MetaClass definition of the class if it has one. This file is included only by the C file which defines the class Xyz functions (xyz.c).
xyz.c	This file holds the function definitions for Xyz. Each function in this file is prefixed with Xyz and takes a pointer to Xyz as its first parameter. These files are included only if you have purchased the source code option.

C+O Source Code Organization

Library Organization

C+O comes with precompiled object code organized into object libraries. Libraries are provided for several different memory models, including small, medium, large and compact.

Libraries are also provided for creating debugging and production executables. This allows you to test your code with the larger, slower debug libraries which will help you catch many bugs before they cause a problem. Then when you are ready to create a so-called production executable, you can link with the production libraries which are much faster and smaller, with all the debugging code removed. The following list describes each library provided.

cobjds.lib	Small memory model. Debug version.
cobjdm.lib	Medium memory model. Debug version.
cobjdl.lib	Large memory model. Debug version.
cobjdc.lib	Compact memory model. Debug version.
cobjps.lib	Small memory model. Production version.
cobjpm.lib	Medium memory model. Production version.
cobjpl.lib	Large memory model. Production version.
cobjpc.lib	Compact memory model. Production version.

Libraries supplied with C+O

Test-Driving Classes

C+O comes with a set of demo programs which allow you to test C+O classes in a controlled environment. You may want to study this code to understand better how to use C+O class libraries.

To run the test programs you must first compile and link them. Batch file procedures exist in the C:\COBJECTS\TST directory for compiling and linking all the test programs. Each test program uses one or more classes. The following table identifies the test programs provided and what they do.

tstblk.c	Demonstrates the Block class.
tstcls.c	Demonstrates the Class class and to a lesser degree Object class.
tstdll.c	Demonstrates the List class. Should also see tstlel.c which demonstrates ListElements.
tstdpa.c	Demonstrates the DynamicArray class. Also shows example of abstract superclass and message sending.
tstgrf.c	Demonstrates the Graph, Vertex and Edge Classes. Also provides a simple example of how an Adventure game might be set up using these classes.
tstjul.c	Demonstrates the JulianTime class.
tstlel.c	Demonstrates the ListElement class. Should also see tstdll.c which demonstrates the List class.
tstmcl.c	Demonstrates the MetaClass and to a lesser degree, the MetaMessage and MetaSuperClass classes.
tstobj.c	Demonstrates the Object class.
tststr.c	Demonstrates the String class.
tsttre.c	Demonstrates the Tree class.
tsttsk.c	Demonstrates the Task Class (mainly exception handlers.) Also see tsttsk1.c, tsttsk2.c, and tsttsk3.c

Demonstration and Example Source Code

This page is intentionally left blank

Writing a Program Using C+O Class Libraries

In this section, you will learn how to write, compile, and link a C+O program. Before beginning, it is recommended that you have already read section Three, *The Object-Oriented Approach to Developing Software.* This section is printed first for those who can't wait to dive in!

Anatomy of a C+O Program

In this section we will take you through the steps needed to compile and link a program using C+O classes.

Let's start by creating the shell of a C+O program. The main thing we want to show here is that all C+O programs need to set up at least one exception handler near the beginning of the program. The reason for this is that C+O may detect a bug while running. Rather than try and handle it locally, it jumps back up to the exception handler which typically prints the exception information and then exits the program (after some cleanup we assume). Thus, even a buggy program can be well-behaved. In general all C+O programs need to be set up in this fashion. Example 1 shows the main() function for a C+O program.

```
#include "cobjects.h"
#ifndef  TSK
#include "tskmac.h"
#endif

static  PSTR     ModuleName = MODULE_NAME;

int     main( int argc, char **argv )
{
  TskDefaultInit( TskMain, argc, argv );

  if ( TskOnException( TskMain ) ) {
    TskPrintException( TskMain );
    return TskExit( TskMain, 1 );
  }

  /* Application logic goes here... */

  TskNormalExit( TskMain );
}
```

Example 1 - Typical main() function for C+O program

Writing C+O Programs

All C+O programs need to include the header file "cobjects.h". This file defines many typedefs used by C+O classes. These typedefs help make C+O libraries more portable than they otherwise would be. C+O libraries do not use any of the built-in types provided by C directly. All fundamental datatypes used by C+O are defined as typedefs in this file. For example, SmallInt, MediumInt, and LargeInt are int typedefs for 8 bit, 16 bit, and 32 bit quantities respectively. String is a char, and PSTR a pointer to char. The table at the beginning of this manual, *C+O Datatypes Table*, explains the different datatypes and keywords defined by C+O.

Next let's look at the code fragment immediately following the inclusion of cobjects.h. This code fragment is how we include an H file in C+O programs.

```
#ifndef  TSK
#include "tskmac.h"
#endif
```

Example 1a - Conditional inclusion of an H file

Each class header file defines a macro when the file is included. Because C+O headers also include class headers for other classes, we would like to make sure we only read in those files once. Hence, we always conditionally include files based on a macro being undefined. The following table shows what naming convention is used for various H files:

xyz.h	XYZ_H
xyzmac.h	XYZ
xyzcls.h	XYZCLS

Table 1b - Conditional include variables

Moving back to our original Example 1, we have included tskmac.h. This file will automatically include tsk.h. We include this file because we will be calling Task functions. In general, whenever we want to access one or more functions that belong to a class, we must include the xxxmac.h header for that class (where xxx is the class prefix).

Next we define the static variable ModuleName. This variable is referenced by assertion macros. MODULE_NAME is the name of the C file being compiled.

```
static  PSTR    ModuleName = MODULE_NAME;
```

Example 1c - Defining ModuleName

Assertion macros are used to check the logic of programs. If the condition specified in an assertion fails, an exception is generated and generally a message displaying the filename and line number where the exception was generated appears. We will get back to exception handlers in a minute. By using a variable instead of a string literal (as Microsoft's implementation of assert does), we do not repeatedly store the same string in the constant data segment of our executable. This can be a significant chunk of memory when you use lots of assertions as C+O does.

Next, after the declaration of main(), we want to initialize a Task object. The Task class automatically declares a global Task instance called TskMain.

```
TskDefaultInit( TskMain, argc, argv );
```

Example 1d - Initializing TskMain

The Task class is used to model the C+O program itself. There is typically only one instance of a Task in a running program. This would be TskMain.

The Task class is important to all C+O programs because it defines the exception handling mechanisms. C+O relies heavily on exception handlers to signal problems in using C+O libraries. C+O is very good at detecting invalid pointers, memory corruptions, and uninitialized data.

Once a Task has been initialized (preferably TskMain), we can set up an exception handler.

```
if ( TskOnException( TskMain ) ) {
   TskPrintException( TskMain );
   return TskExit( TskMain, 1 );
}
```

Example 1e - Setting up an exception handler

An exception handler is set up by calling TskOnException. The return value from TskOnException is 0 when the handler is set. However, if an exception is raised, even if it is several layers of function calls below main, then control is transferred to the statements inside the if statement. It is then up to these statements (the exception handler) to determine how to process the error. TskOnException behaves very much like the Ansi C function setjmp() and in fact is implemented using setjmp().

The particular exception handler shown here will print a message displaying the filename and line number where the exception was detected and then exit the program.

Finally, the statements following the exception handler are the statement which make up your program. When the program is ready to exit it should call TskNormalExit(). Please take a look at the Task Class Summary for a more complete discussion of the functions discussed here and others which will be of interest. Also, be sure and look at the example Task code provided. It demonstrates exactly how exception handlers behave under a variety of circumstances.

Creating a Class of Your Own

To use some C+O classes, you must build a new class on top of an existing one. These classes include: ListElement, List, Tree, Vertex, Edge and Graph. For more information on how to do this, read section Three, *The Object-Oriented Approach to Developing Software*, and also study the related example code.

Other classes, specifically JulianTime, String, DynamicArray, and Memory are usable as is without prior setup. However, the principles of encapsulation still apply and you may wish to create a class which incorporates one or more of these classes.

Finally, several classes exist for supporting the object-oriented functionality of C+O classes and for supporting exception handling. These classes include: MetaClass, MetaMesage, MetaSuperClass, Class, Object, Message, Block, and Task. You will be using some functions from almost every one of these classes in building a C+O program so it is important you understand their role and usage. It is doubtful you will want to extend these classes, but should you wish to, please try and keep your extensions upwardly compatible -- after all, these classes provide the foundation for object-oriented programming in all C+O class libraries.

For examples of using classes, please refer to the example code provided. A separate program is provided for most classes.

Because many C+O functions are macros, you should avoid calling functions with parameters which have side effects such as the pre or post increment operators.

Debug and Production Libraries

C+O provides two kinds of libraries, debug and production. The major difference between them is that the debug library includes lots of argument checking whereas the production library does not. Another difference is that the production libraries use C+O macros and the debug libraries use the function equivalent.

When you are first testing a program that uses C+O classes you will want to compile and link it with the debug libraries so as to catch any error of use. When you have debugged your program, you can recompile it for the production libraries and the resulting executable will be much smaller and much faster.

You must always compile your code appropriately for the particular C+O library you intend to link with. This means you must choose an appropriate memory model and you must define the COBJ_PRD flag (preferably with compiler startup flags) if you are going to link with the production libraries. Structure sizes are different for debug and production versions so it is imperative that you not mix these two styles.

Compiling

Compiling a C+O program is the same as compiling any other. You may use whatever compiler switches you want but you must be aware of the following things.

First, the C+O libraries are compiled without packing structure members (/Zp option). Compiling your code with this option will cause problems.

C+O class libraries are compiled for the alternate floating point math package.

You must define the identifier COBJ_PRD when you wish to produce your final production executable. If you are intending to link with a debug library, do not define this switch. In either case, your entire program must be compiled with or without this flag defined. The following example shows a compile line for compiling for debug.

```
C:>CL  /D COBJ_PRD /AL /W3 /Ox /Fpa /C /I C:\COBJECTS\INC XYZ.C
```

How to Compile for Production Code, Large Model

The following example shows the same compile line, but this time generates production code.

```
C:>CL  /AL /W3 /Ox /Fpa /C /I C:\COBJECTS\INC XYZ.C
```

How to Compile for Debug Code, Large Model

Should you need C+O libraries compiled with other options, memory models, or floating point packages, please contact Objective Systems. We can provide you with these libraries at little or no cost.

Writing C+O Programs

Linking

You may link C+O programs with the standard linker provided by Microsoft. If you have compiled your code with the preprocessor variable COBJ_PRD, you must link with the production libraries (COBJPx.LIB where x is the memory model letter). Otherwise link with the production libraries (COBJDx.LIB). The following example shows how to link a C+O program compiled for debug and the large memory model.

```
C:>LINK  XYZ,XYZ,XYZ,\COBJECTS\LIB\COBJDL \MSC\LIB\LLIBCE;
```

Debugging

You may use whatever debugger you are using now to debug C+O programs. Codeview is our preference. C+O libraries are not compiled with any debug information. If you have the source code option, you may of course recompile them with this option.

We recommend compiling and linking with our debug libraries when you are first testing a C+O program. The C+O debug libraries trap many bugs before they can cause a problem. When a bug is detected an exception is generated, causing an exception handler to be invoked. This will typically result in the program displaying the filename and line number where the exception was detected. You will want to use Appendix A to determine the cause of the exception.

Because many C+O functions are macros, you should avoid calling functions with parameters which have side effects such as the pre or post increment operators.

Optimizing

Optimizing your C+O program is as simple as recompiling it with the production compiler flag set and linking with the production libraries. Many C+O functions are written as macros and the production compile flag COBJ_PRD enables these macros. Most or all of the parameter checking to C+O functions is disabled as well when this flag is set.

Because many C+O functions are macros, you should avoid calling functions with parameters which have side effects such as the pre or post increment operators.

The Object-Oriented Approach to Developing Software

Introduction

Much has been written lately about object-oriented programming (OOP) and object-oriented programming languages (OOPLs). When we think of object-oriented programming, we probably think of SmallTalk, C++ or LOOPS. However, all of the concepts rigorously applied in those languages can be applied (with a little extra effort) to commercial languages such as C. And by applying those concepts, we can attain the productivity benefits promised by advocates of object-oriented programming while retaining the benefits of programming in C.

C+O (Classes + Objects) class libraries are the first commercial products designed to apply object-oriented programming techniques specifically for the C language without requiring a pre-processor or interpreter. C+O class libraries are efficient and can be used to deliver commercial quality products. C+O is a tool for increasing your productivity by decreasing the amount of general purpose code you must write (and debug) for every program. It is also a tool for allowing you to create your own libraries of reusable code. C+O has extensive debugging facilities for helping you track down problems.

Of course, you may already be building "black box" code. Code which is insulated from other parts of a program and has a very well defined interface. In this case, you already will be familiar with some of the concepts provided by C+O class libraries. C+O offers a standard way of designing and building such black boxes. Its *comprehensive* and rich set of tools and engineering techniques allow you to build more flexible black boxes than ever before.

In this section, you will learn the fundamentals of how to create and use reusable code with C+O class libraries. The examples presented here are all coded using C+O stylistic and naming conventions. After reading this section and working through the examples, you will have the basic knowledge needed to apply C+O object-oriented programming concepts to real problems, taking advantage of C+O reusable code libraries.

You may want to look at and run the demo code provided with C+O as a means of further enhancing your understanding of the concepts presented here.

The Object-Oriented Approach

Background

The general misconception regarding object-oriented programming is that it is of use only for designing user interfaces or graphics packages. However, object-oriented programming techniques are beneficial for building any kind of software. Benefits include faster implementation (due to reusability), greater reliability (due to reusability) and better maintainability (due to encapsulation and dynamic binding). In short, object-oriented programming, or OOP is about finding better ways to modularize software while making it more reusable.

Part of the problem in getting the word out about OOP is that to date, most of the work done involving OOP was written in Smalltalk. The reason that this is a problem, is because the terminology associated with Smalltalk is completely foreign to the terminology we are accustomed to from using commercial languages. The result is that when someone from the object-oriented community tries to explain to us the workings of object-oriented programming, it's very much like a first lesson in French -- the teacher stands up there talking away and we're sitting there wondering what's going on.

One other problem is the common perception that object-oriented programming implies high-overhead programming. Smalltalk has done much to contribute to that perception. It consistently applies OOP techniques to solving problems with about the same finesse as using a .44 magnum to kill a fly. To keep OOP a tool we can use efficiently, we must have an arsenal of OOP techniques equivalent in range from flyswatters to cannons -- and we must have the wisdom to know which tool is right for the job.

What makes C+O different from Smalltalk and many other object-oriented languages and extensions, is that first, we demystify the language of object-oriented programming and place it firmly back in the hands of the people building commercial products. Second, we provide a range of object-oriented solutions which keep the amount of program overhead and program complexity in line with the benefits they produce. In this way, you can apply the object-oriented solutions which best fit your particular problem.

Before moving on, we should mention how C++ fits together with C+O class libraries. First of all, the benefit to using C+O class libraries is not the fact that it is object-oriented. The benefit to using C+O class libraries is that you decrease your programming effort by reusing its code. It is therefore our intention to provide C+O class libraries for standard-setting languages. If C++ becomes something of a standard, as we expect it will, we will port C+O to that language. When we do, we will fully adopt the object-oriented paradigms provided by that language. Currently however, C++ is still somewhat in a state of flux and their are very few vendors providing C++ compilers/pre-processors.

C+O was developed by Objective Systems over the last two years. It was developed specifically to allow us to create highly reusable code which was efficient enough to be used in commercial product development.

Classes

In explaining the object-oriented precepts behind C+O, we will always keep the discussion centered around the C language. We will borrow many terms from the OOPL vocabulary but define them in terms which a C programmer can quickly comprehend. It is therefore a prerequisite that you have a solid knowledge of C programming. It is not a prerequisite to have a knowledge of object-oriented programming although it will help if you have some basic familiarity with the subject.

The first thing to consider are **classes**. A class is the fundamental unit of programming in object-oriented languages and we adopt the concept for C+O. Example 1 shows the class Point.

```
struct  Point {
    MediumInt  x;
    MediumInt  y;
};
typedef     struct Point        Pnt;
#define     PPNT                struct Point *

Void        PntSetXY( PPNT, MediumInt, MediumInt );
MediumInt   PntGetX( PPNT );
MediumInt   PntGetY( PPNT );

#ifdef      PNT_MACROS
#define     PntGetX( pPnt ) ((pPnt)->x)
#define     PntGetX( pPnt ) ((pPnt)->x)
#endif
```

Example 1 - Point Class

A class is defined as a structure and functions which manipulate and access that structure. Another word we use interchangeably with function is **method**. Only the functions (methods) associated with a class can access the structure members directly. Functions not belonging to a class must rely on the functions provided by that class for manipulating and accessing the structure. If we could directly represent this in C, it would be as if functions were always declared as part of a struct declaration. Instead, we rely on source code organization and naming conventions to help reiterate the relationship. This property of not allowing direct access to structures other than through methods belonging to the class is called **encapsulation**.

The Object-Oriented Approach

Referring back to Example 1, we show how the Point class would be defined in C+O. The Point class consists of a struct declaration for Point and some functions with the names prefixed by Pnt. Sometimes we define macros for simple functions. We also define a typedef for Point called Pnt (always a three character abbreviation) and a #define for a pointer to struct Point called PPNT. C+O uses this naming convention consistently throughout its classes. Finally, we always require that the first parameter to a Point function a pointer to Point. From the perspective of source code organization, we would declare the Point structure in a file called "pnt.h", the function declarations and (optional) macros in "pntmac.h", and the function definitions in "pnt.c". This makes it very simple to locate the definition or declaration of an item based on its prefix.

You have now learned the single most important concept used in object-oriented programming. Simply, that programs are structured into classes. Notice there is no magic in this, it is simply a design decision which reaps certain benefits. Namely, it makes programs more modular and in general easier to understand. Programs in which functions directly access the implementation of more than one structure are inherently less modular and the code less reusable. Another benefit of classes is that it is much easier to change the implementation of a class without affecting a potentially enormous and unknown amount of code. As long as the external interface (and semantics) provided remains the same, other classes will not be affected.

Of course, there is nothing preventing a C programmer from violating the class concept (unlike most OOPLs). C won't even warn you that a class is being violated. It is a matter of discipline. You must believe that its important not to violate the class rule and that certain benefits are provided by doing so. While we believe you will get more out of C+O by following this convention, it is up to the individual programmer to decide.

Before continuing, it is important to note one more thing, namely, that there is an important psychological benefit to the class concept. Namely, it gets us thinking about building a program in terms of *parts*. Once we get into this mind-set, it is natural for us to want to reuse those parts in other programs or even within the same program. This is at the heart of what C+O is about.

Now that we understand what a class is, we will interchangeably refer to it as a structure, in deference to our C background, but keeping in mind that we still associate with it a particular set of functions for manipulating that structure.

Objects

A structure declaration has no existence of its own in a running program. Only when we declare variables of some structure type can we assign or retrieve values from the structure. We must also name the particular variable since there may be more than one variable declared with that same type. We refer to the memory allocated to a variable, whether it be external, static, automatic, or dynamic, as being an **instance** of that structure. We also refer to instances as **objects**. Example 2 shows some Point objects.

```
Point   x;      /* An instance of Point class named x */
Pnt     y;      /* A Point object named y */
PPNT    pPnt;   /* Pointer to a Point object named pPnt */
```

Example 2 - Point Objects

This is the second most important thing to understand about object-oriented programming. Namely, that while our program is structured in terms of classes, when it is running, it is objects which we are manipulating.

Let's continue with the Point example and fill in the functions for manipulating Points. Example 3 shows two of the Point methods being defined. In C+O, methods always manipulate pointers to objects of the class.

```
Void    PntSetXY( PPNT pPnt, MediumInt x, MediumInt y )
{
    pPnt->x = x;
    pPnt->y = y;
}

MediumInt  PntGetX( PPNT pPnt )
{
    return pPnt->x;
}
```

Example 3 - Point Methods

We have now seen a complete class definition. Our example shows only a small number of functions for class Point. This is how most classes start out. Over time, you and others will add more functions, rounding out a class implementation.

Designing with Class

One of the interesting things about classes, is how they force us to think about where to add a function, what types of parameters to pass, how to name it, what effects it will have, and so on. In other words, classes provide a structure for thinking about how we design and organize code. This shift of thinking from traditional programming tends to inspire better function and program design.

A couple of words are in order regarding good class design and good design of functions belonging to a class. First, the class structure should reflect a single entity, or a single relationship. For instance a spreadsheet program would have classes called Cell, Formula, and Spreadsheet, among others. If you find yourself designing a class which ends up seeming like two or more things, break it apart! Any small increase in overhead will be well worth the resulting simplified program architecture. Look for application specific classes which can be broken into a non-application specific part and an application specific part. The former you may be able to reuse in another program.

When designing functions, have lots of little, simple functions which perform a single act, rather than large functions which do many tasks. If you name your functions according to the task the perform, as we do in C+O, you will find that overly complex functions cannot be named in under 32 characters or that the names become rather bizarre, sounding more like a run-on sentence than a verb. This advice has saved us from bad design decisions on many occasions.

Another good rule of thumb are functions which take large numbers of parameters. These are often suspect of being amenable to decomposition.

Finally, and we cannot overstate this, avoid writing functions which rely on data other than the parameters being passed into the function. In other words, minimize or eliminate global variables. One way to think about this is what if you had to pass every global variable you accessed in your function as a parameter instead. Would it seem unwieldy? If so, then you've got a function which is side-effect heavy.

Foundation

Everything we are going to tell you from here on out concerns two related things: how to build classes on top of other classes and how to make instances of those classes communicate with one another. You can think of this as the glue which binds classes together or the grease which keeps them from grinding against each other. Both analogies are relevant.

The mechanisms which allow objects to communicate are crucially important to the success of an object-oriented system. The reasons are this: if communication between objects is slow, then the system is doomed to be slow. If communication mechanisms are limited, then the flexibility of the system is compromised. C+O uses a variety of mechanisms to allow objects to communicate. None of them are "slow".

The mechanisms which allow classes to be built on top of other classes are also extremely important. For instance, if you can only build a class based on one other class (as many OOPLs provide), you are limited to making more and more refined versions of the same class. On the other hand, if you can build new classes from any number of other classes as C+O provides, you have a much more powerful system.

Example 4 shows how a class can be built from other classes.

```
struct Rectangle {
    Point   origin;
    Point   extent;
};
typedef  struct Rectangle  ;
typedef  struct Rectangle  Rct;
#define  PRCT  struct Rectangle *;

Void     RctSetOriginAndExtent( PRCT, PPNT, PPNT );
Void     RctGetSizeX( PRCT );
Void     RctGetSizeY( PRCT );
Void     RctGetOrigin( PRCT, PPNT );

Rct      R = {0};
```

Example 4 - Rectangle Inherits from Point Twice

In its simplest form, building a class from another class is simply a matter of including the structure of one inside the other. Referring to Example 4, we say that the Rectangle class **inherits** from the Point class. When a class includes

This example immediately begs the question of how does one communicate between instances of classes. Using Example 4, the Rectangle R can be thought of as actually being three objects: the Rectangle R, the Point R.origin, and the Point R.extent.

First, because Point is defined independently of Rectangle or any other class, there is no way R.origin and R.extent can communicate directly. Similarly, neither Point object can communicate with R since they are unaware of R as well.

Therefore, it is up to R to communicate to with both Points. Thus we have what we call **simple inheritance**. Put another way, Point has no mechanisms or functions for communicating with structures which include it as a struct member.

Before we go any further, we need to define some terminology for the relationship of Rectangle to Point and Point to Rectangle. We call Rectangle the **subclass** of Point and Point (each one) the **superclass** of Rectangle. This terminology is carried over (and extended) from Smalltalk.

These terms may seem reversed and contrary to common sense. The usage seems to be derived from the fact that in Smalltalk there is only one superclass which a class can be built upon. Thus a subclass is viewed as being a more restricted form of its superclass. Multiple-inheritance certainly changes this perspective, however we will stick by Smalltalk's definition of these terms rather than make matters worse by reversing them. In any case, we prefer to call Rectangle the **client** of Point and Point a **server** to Rectangle.

We need to make clear that a structure cannot inherit from a structure which it points to, only one which is included as part of its definition. Thus, if Rectangle were defined as in Example 5, it would not be considered to inherit from Point. The reason for this is that inheritance is defined as being a contiguous extension of one class instance from another. There can only be one instance of a subclass for an instance of a superclass. In Example 5, changing the pointer values of origin or extent would be to change the particular instances they inherit from. To allow otherwise would invite chaos.

```
struct Rectangle {
    Point    *origin;
    Point    *extent;
};
```

Example 5 - Rectangle Does Not Inherit From Point

Unlike some object-oriented languages, we take the view that a subclass may not "look inside" its superclass(es). In our example, Rectangle R may not directly access the contents of either Point (origin or extent). Rectangle R must use the functions provided by the Point class when accessing R.origin or R.extent. Of course there is no way for the compiler to enforce this, you must simply believe this rule important enough to follow. The rationale for this decision is simply that we don't believe there is any benefit to allowing a client to have direct access to its servers. The black box approach is much safer.

Example 6a shows the wrong way for Rectangle to communicate with Point. Example 6b shows the correct way.

```
MediumInt  RctGetSizeX( PRCT pRct )
{
    return pRct->extent.x - pRct->origin.x;
}
```

Example 6a - Violates Encapsulation Rule

```
MediumInt  RctGetSizeX( PRCT pRct )
{
    return PntGetX( &pRct->extent ) - PntGetX( &pRct->origin );
}
```

Example 6b - Encapsulation Rule Satisfied

Finally, Example 6c goes one step further and creates a new Point function PntGetDiffX. This is in the spirit of creating functions which may be likely be reused somewhere else. One must always keep in mind that every loop, every calculation, every process defined may be of value to you or someone else at a later time. Coding the function inside of a different class helps no one.

```
MediumInt  PntGetDiffX( PPNT pPnt1, PPNT pPnt2 )
{
    return pPnt1->x - pPnt2->x;
}

MediumInt  RctGetSizeX( PRCT pRct )
{
    return PntGetDiffX( &pRct->extent, &pRct->origin );
}
```

Example 6c - Maximizes Reusability by Creating New Point Method

Advanced Inheritance

More complicated structures can be created which require more sophisticated styles of interplay than previously described. Example 7 shows the classes Node and BinaryTree (for a complete discussion of binary trees, see Knuth, The Art of Computer Programming, Volume 1).

```
#define  PBIN  struct BinaryTree *
#define  PNOD  struct Node *

struct BinaryTree {
     Object  obj;
     PBIN    left;
     PBIN    right;
};
typedef  struct BinaryTree  BinaryTree;
typedef  struct BinaryTree  Bin;


struct Node {
     Object  obj;
     PSTR    name;
     BinaryTree      bin;
};
typedef  struct Node  Node;
typedef  struct Node  Nod;
```

Example 7 - Node and Binary Tree Classes

A binary tree has left and right pointers to other binary trees. The left subtree is shown as being connected to its branch with a vertical arrow, the right subtree is shown as being connected to its branch by a horizontal arrow.

In Example 7, Node is inheriting BinaryTree. The inclusion of Object structure members in both classes signifies a more complex form of inheritance than we have encountered before. It is a requirement for this type of inheritance that the Object structure members be the first member of the structure. It is the Object class which gives other classes their object-oriented characteristics.

Example 8 shows how we create new instances of type Node with the function _NodNew. We prefix the function name with an underscore to easily identify it as a Node function which does not take a Node pointer as its first argument. The function BinInit is used to initialize an instance of Bin and should always be the first Bin function called after allocating memory for the object.

```
Void    BinInit( PBIN pBin )
{
    pBin->left = pBin->right = NULL;
}

PNOD  _NodNew( PSTR name )
{
    PNOD  pNod;

    pNod = (PNOD) ClsNewObject( NodCls );
    BinInit( &pNod->bin );
    pNod->name = malloc( strlen( name ) + 1 );
    cpystr( pNod->name, name );
}

{
    a = _NodNew( "a" ); b = _NodNew( "b" );
    c = _NodNew( "c" ); d = _NodNew( "d" );
    e = _NodNew( "e" ); f  = _NodNew( "f" );
    g = _NodNew( "g" ); h = _NodNew( "h" );
}
```

Example 8 - Creating Nodes

The variable NodCls (referenced in _NodNew) is a pointer to an object of type Class and is initialized soon after the program starts up. Class objects are used to create and initialize instance of classes in C+O. Later we will show you how to create the Class instance NodCls. The function ClsNewObject allocates memory for an object as specified by NodCls.

Now that we've created some Nodes, we need functions to manipulate them. Example 9 shows some BinaryTree methods for connecting up BinaryTrees to the left and right subtrees. It also shows the equivalent Node functions. In effect, we are demonstrating how Node is inheriting some behavior (functions) from BinaryTree. The Node functions NodSetLeft and NodSetRight would typically be defined as macros for efficiency.

```
Void    BinSetLeft( PBIN pBin, PBIN pBinLeft )
{
    pBin->left = pBinLeft;
}

Void    BinSetRight( PBIN pBin, PBIN pBinRight )
{
    pBin->right = pBinRight;
}

Void    NodSetLeft( PNOD pNod, PNOD pNodLeft )
{
    BinSetLeft( &pNod->bin, &pNodLeft->bin );
}

Void    NodSetRight( PNOD pNod, PNOD pNodR )
{
    BinSetRight( &pNod->bin, &pNodR->bin );
}

{
    NodSetRight( a, b );
    NodSetRight( b, c );
    NodSetLeft( a, d );
    NodSetLeft( d, e );
    NodSetRight( e, f );
    NodSetLeft( c, g );
}
```

Example 9 - Node Inherits Behavior From BinaryTree

For Node to communicate with BinaryTree is a relatively trivial matter as we have seen. But now, we need a way for BinaryTree to communicate back to Node if we are to use Binary Tree access methods. Example 10 shows the problem.

```
PBIN    BinGetLeft( PBIN pBin )
{
    return pBin->left;
}

PNOD  NodGetLeft( PNOD pNod )
{
    return BinGetLeft( &pNod->bin, NodOffset );
}
```

Example 10 - Wrong Way to Implement NodGetLeft

In Example 10, BinGetLeft returns a pointer to a BinaryTree, not a Node. Therefore the implementation of NodGetLeft is incorrect, it is returning a BinaryTree pointer instead of a Node pointer. What we need is a mechanism by which Bin can return to us Node pointers. Example 11 shows how we do this with C+O.

```
POBJ   BinClientGetLeft( PBIN pBin, MediumInt offset )
{
    return ObjGetClientOrNull(&pBin->left->obj,offset);
}


PNOD  NodGetLeft( PNOD pNod )
{
    return (PNOD) BinClientGetLeft( &pNod->bin, NodOffset );
}
```

Example 11 - Simple Communication From BinaryTree to Node.

What's going on here? Simply that BinaryTree is storing BinaryTree pointers (left and right). But because each BinaryTree is a superclass of Node (we assume a homogeneous BinaryTree), we can compute the Node pointer from a BinaryTree pointer by adding an offset to the BinaryTree pointer. The function ObjGetClient does just that. It adds the (negative) offset to the object pointer to yield a new object pointer. Note that we must cast the Object pointer returned by BinClientGetLeft to a Node pointer. This is why the Object superclass must always be the first member of a structure.

Table 1 shows the memory layout for the objects of class Node and BinaryTree. Each structure member is of a fixed size and a fixed offset. Therefore, if we know the address of a structure member, we can compute the address of the structure which includes it. (Sizes shown are for the small memory model.)

Class	Size	Mem	Size	Offset	Description
Nod	10	obj	2	0	
		name	2	2	
		bin	6	4	The value of NodOffset
Bin	6	obj	2	0	
		left	2	2	
		right	2	4	

Table 1 - Memory Layout for Node and BinaryTree Objects

The offset value NodOffset gets computed automatically in the early stages of the program (something we will describe later). On first glance, having to pass in offsets may seem a bit of a nuisance. Keep in mind though that BinaryTree is being written independently of Node. We would like to build other classes on top of BinaryTree, and they will have different offsets.

C+O has facilities for allowing us to ignore offsets (with no additional cost in complexity or storage), but we typically would not do this. The reason is because offsets have a performance advantage when multiple levels of inheritance occurs, which typically is quite often. For example, we might want to allow Node to be inherited by another class. This means we need to add a (negative) offset to Node pointers to get client structure pointers. Example 12 shows how this works in practice.

```
POBJ   NodClientGetLeft( PNOD pNod, MediumInt offset )
{
    return BinClientGetLeft( &pNod->bin, NodOffset + offset );
}
```

Example 12 - Inheritable Node routine

In Example 12, to get the client structure pointer of the left subnode of pNod, we get the left subtree of the BinaryTree pNod.bin, add NodOffset to get the left subnode of pNod, then add offset to get the client structure of that subnode. When these type of functions get converted into macros, the resulting code is very efficient.

Visiting Objects

We now understand how a class can communicate with its client, albeit in an indirect way. Now let's see how a class can call a client function. Example 13 shows how this can be done.

```
Void BinClientVisitPreOrder(PBIN pBin,MediumInt offset,PBLK pBlk)
{
    PBIN left;
    PBIN right;

    if ( !pBin )
        return;
    left = pBin->left;
    right = pBin->right;
    BlkExecute( pBlk, ObjGetClient( pBin, offset ) );
    BinClientVisitPreOrder( left, offset, pBlk );
    BinClientVisitPreOrder( right, offset, pBlk );
}

Void    NodVisitPreOrder( PNOD pNod, PBLK pBlk )
{
    BinClientVisitPreOrder( &pNod->bin, NodOffset, pBlk );
}

Void    NodPrintPreOrder( PNOD pNod )
{
    Block   blk;

    BlkInit( &blk, NodPrint );
    NodVisitPreOrder( pNod, &blk );
    BlkDeInit( &blk );
}

Void    NodPrint( PNOD pNod )
{
    printf( "Node is %s\n", pNod->name );
}
```

Example 13 - Node Visit Function

Before we talk about what the BinClientVisitPreOrder and NodVisitPreOrder function are doing, we need to introduce a new class called Block. A Block object can store a function pointer and several arguments. At some later time, a call to BlkExecute will cause the stored procedure to be executed and any parameters stored in the Block will be passed along to it. The first parameter sent to the function is a pointer which is passed in at the time BlkExecute is called. BlkExecute behaves much like an indirect function call.

Ok, now back to BinClientVisitPreOrder and NodVisitPreOrder. These are what we call **traversal** functions. A traverse function is said to **walk** the nodes of a data structure in some predefined order. As we walk the data structure, we **visit** each node along the way. (We use the term node generically and Node to refer to the Node class).

In Example 13 what happens is the following. BinClientVisitPreOrder walks the binary tree in pre-order. (Pre-order is just one of many ways to visit all the nodes of a binary tree.) To visit a node, it calls the function BlkExecute passing it the client of the BinaryTree node being visited. BlkExecute will call the client function (established by Node) passing the object pointer it receives as a parameter. In this way, BinClientVisitPreOrder is calling (visiting) a Node function and passing it a Node pointer for each BinaryTree.

Sending Messages

In our next example, we show how a client can inherit behavior through messages from a server class as well as override default behavior for a server class.

In Example 14a and 14b we show two different methods for deallocating nodes of a BinaryTree. Only one of them is correct. In both cases, the function ObjDestroy function is called to deallocate the object. ObjDestroy is always used to deallocate objects created by ClsNewObject.

In the incorrect one, Example 14a, BinDestroy tries to free BinaryTree objects. However, the BinaryTree object is part of a larger object that was allocated, namely a Node. Therefore, the call to ObjDestroy will not succeed (it can tell if there is a subobject). Worse, even if that could work, the Node has additional allocated memory which must be reclaimed prior to the Node being freed.

```
Void    BinDestroyAll( PBIN pBin )
{
    Block   blk;

    BlkInit( &blk, BinDestroy );
    BinClientVisitPreOrder( pBin, 0, &blk );
    BlkDeInit( &blk );
}

Void    BinDestroy( PBIN pBin )
{
    ObjDestroy( &pBin->obj ); /* This is an error */
}
```

Example 14a - Wrong Way to Free Nodes

Example 14b shows how a Node function could deallocate itself and all its subnodes. But is there any way we could write BinDestroyAll so that it works properly?

```
Void    NodDestroyAll( PNOD pNod )
{
    Block   blk;

    BlkInit( &blk, NodDestroy );
    NodVisitPreOrder( pNod, &blk );
    BlkDeInit( &blk );
}

Void    NodDestroy( PNOD pNod )
{
    free( pNod->name );
    ObjDestroy( &pNod->obj );
}
```

Example 14b - How to Destroy All the Nodes in a BinaryTree

The answer is yes. The solution is to have the BinaryTree object send a message requesting that the object be freed. Sending a message is like calling a function except you're not sure the exact function which will be called. This may sound confusing. However, it is up to the function which gets invoked to make sure the action you requested takes place. This process of choosing the function at run-time rather than at compile time is called dynamic binding.

By default, if Binarytree has no subclass, the destroy message (MSG_BIN_DESTROY) will call BinDestroy. But Node **overrides** this message with its own function NodDestroy which does the appropriate cleanup and frees the node. (We are just about ready to show you how all this is done, so hang on.) Example 15 shows the code which implements this properly.

```
Void    BinDestroyAll( PBIN pBin )
{
    Block   blk;

    BlkInit( &blk, BinSendDestroy );
    BinClientVisitPreOrder( pBin, 0, &blk );
    BlkDeInit( &blk );
}

#define     MSG_BIN_DESTROY    0

Void    BinSendDestroy( PBIN pBin )
{
    Block   blk;

    BlkInit( &blk, NULL );
    ObjSendMessage( &pBin->obj, MSG_BIN_DESTROY, pBlk );
    BlkDeInit( &blk );
}
```

Example 15 - Sending a Message

This demonstrates how an object's default behavior can be changed by a client class without changing the code for the server class. This extends the ability to make reusable classes even more general and feature rich than they otherwise could be.

MetaClass

At this point we need to explain how Object and Class classes understand so much about structures and offsets and how they can send messages to one another. C+O provides a number of classes which allow this behavior to be defined. The first class to examine is MetaClass.

The MetaClass class is used to define the structure of a class, characteristics of objects of that structure type, superclasses of that class, and messages which it inherits and may be inherited by other classes. Example 16a shows the MetaClass definition for BinaryTree. You may wish to refer to the reference sections on MetaClass, MetaMessage, and MetaSuperClass for a complete description of the structure members of these classes.

```
/* MetaMessages for class BinaryTree */
Mms  BinMms[] = {
  { MMS_CHECK_WORD,      /* Default value */
    NULL,                /* Superclass msg to override */
    "destroy",           /* Name of message */
    BinDestroy           /* Default method for message */
  }
};

/* MetaClass instance for class BinaryTree */
Mcl  BinMcl = {
    MCL_CHECK_WORD,      /* Default value */
    "BinaryTree",        /* Name of class being described */
    sizeof(Cls),         /* Default value */
    sizeof(Bin),         /* Size of class instance */
    BinMms,              /* Array of MetaMessages */
    1,                   /* Number of messages */
    NULL,                /* NULL if no superclasses */
    0,                   /* Number of superclasses */
    MclCreateCls,        /* Default class creation func */
    ClsDestroy           /* Default class destruct func */
};
```

Example 16a - MetaClass Definition for BinaryTree

A MetaClass consists of three components: MetaSuperClass definitions, a MetaMessage definitions, and a MetaClass definition (which references the MetaMessage and MetaSuperClass definitions).

In Example 16a, BinaryTree has no superclasses (other than Object which is always assumed), therefore no MetaSuperClass definition appears.

The MetaMessage definition (BinMms) defines one message called destroy for the BinaryTree class. The default method which will be executed when this message is sent is BinDestroy. However, a subclass may (and should in this example) override that message as we will see shortly. The index of the MetaMessage in BinMms is how you identify a message when a BinaryTree object wishes to send a message. That is why MSG_BIN_DESTROY had the value 0 in Example 15b.

The MetaClass definition (BinMcl) defines the size of an instance of type BinaryTree. The other values given are defaults (see MetaClass reference section for more details).

Instances of MetaClass, MetaMessage, and MetaSuperClass (not shown above) are always defined statically. There should be one instance of a MetaClass (and associated MetaMessages, and MetaSuperClasses) for each class you define. Primitive classes need not have a MetaClass as they never communicate with their subclasses.

Example 16b shows the MetaClass definition for the class Node. In it we see that it has one superclass, BinaryTree and overrides one message, "destroy".

```
Node    NodDummy = {0};        /* Example Node instance */
MediumInt  NodOffset = 0;      /* Offset of Bin superclass */

/* MetaSuperClasses for class Node */
Msc  NodMsc[] =
{
  { MSC_CHECK_WORD,            /* Default value */
    "Bin",                     /* Name of superclass instance */
    (POBJ) &NodDummy,          /* Example instance of Node */
    (POBJ) &NodDummy.bin,      /* Example instance of superclass */
    &NodOffset                 /* Store superclass offset here */
  }
}

/* MetaMessages for class Node */
Mms  NodMms[] = {
  { MMS_CHECK_WORD,            /* Default value */
    "Bin",                     /* Override Bin destroy */
    "destroy",                 /* Name of message */
    NodDestroy                 /* Method to override message */
  }
};

/* MetaClass instance for class Node */
Mcl  NodMcl = {
    MCL_CHECK_WORD,            /* Default value */
    "Node",                    /* Name of class being described */
    sizeof(Cls),               /* Default value */
    sizeof(Nod),               /* Size of class instance */
    NodMms,                    /* Array of MetaMessages */
    1,                         /* Number of messages */
    NodMsc,                    /* Array of MetaSuperClasses */
    1,                         /* Number of superclasses */
    MclCreateCls,              /* Default class creation func */
    ClsDestroy                 /* Default class destruct func */
};
```

Example 16b - MetaClass Definition for Node

The variable NodDummy is used in the initialization of the MetaSuperClass instance. It is used to calculate the offset of the BinaryTree superclass from the beginning of the Node structure. This offset is computed when MclCreateClass is called (for NodMcl) and the resulting value is stored in NodOffset.

By overriding the message "destroy", an ObjSendMessage call by an instance of BinaryTree will result in a call to NodDestroy, passing it not a Bin pointer, but a Nod pointer. If Node were to be subclassed by another structure and Node's destroy message was overridden, then that function would be called with a pointer to that class instance. This can go on for as many levels as need be.

During program initialization, we must call MclCreateClass, passing it an instance of a MetaClass. It creates and returns a pointer to an object of type Class. Example 17 shows how this occurs.

```
Class   *NodCls = {0};

    NodCls = MclSendCreateClass( &NodMcl );
```

Example 17 - Creating a Class Instance for Node

Objects of type class are used to create instances of a specific structure. In this case, NodCls can be used to create objects of type Node as we saw in Example 8.

Remember, we only need to define MetaClass definitions and Class instances for class types which we wish to communicate with subclasses or when we are inheriting classes which have MetaClass definitions. Primitive classes, like our Point and Rectangle examples, do not need such definitions. C+O classes which do not need MetaClass definitions are labeled as a Primitive Class in their class reference section.

Frameworks

Next we show how a class can define a **framework** for subclasses to follow. A framework is also known as an **abstract superclass**.

Example 18 shows a class named DisplayShape. DisplayShape implements the basic behavior common to all types of display shapes. It is like any other class with the following exception. It defines messages which are not implemented by the class and therefore must be provided by a client. This is signified by making the method structure member of a MetaMessage be NULL.

```
#define  PDSH  struct DisplayShape *
struct DisplayShape {
     Object  obj;
     Point    mbr;
};
typedef  struct DisplayShape  DisplayShape;
typedef  struct DisplayShape  Dsh;


Mms DshMms[] = {  /* Messages for class DisplayShape */
  { MMS_CHECK_WORD, NULL,  "destroy",  NULL },
  { MMS_CHECK_WORD, NULL,  "display",  NULL },
  { MMS_CHECK_WORD, NULL,  "erase",    NULL },
  { MMS_CHECK_WORD, NULL,  "rotate90", NULL }
};

Mcl  DshMcl = {
     MCL_CHECK_WORD,
     "DisplayShape",
     sizeof(Class),
     sizeof(DisplayShape),
     DshMms,
     sizeof(DshMms)/sizeof(Mms),
     NULL,
     0,
     MclCreateClass,
     ClsDestroy
};
```

Example 18 - DisplayShape Framework Class

In general, and DisplayShape in particular, you cannot create instances of a framework. Instead, you create instances of clients (subclasses) of frameworks.

In Example 19, we define a client class of DisplayShape called RectangleShape. RectangleShape inherits from DisplayShape and provides the display shape - specific functions display, erase, and rotate90. These functions must be rewritten by each class which inherits from DisplayShape.

```
#define PRSH struct RectangleShape *
struct RectangleShape {
    Object obj;
    Rectangle rct;
    DisplayShape dsh;
};
typedef struct RectangleShape RectangleShape;
typedef struct RectangleShape Rsh;

Rsh RshDummy = {0};
MediumInt RshOffset = 0; /* Holds Dsh offset */

Msc RshMsc[] = {
{ MSC_CHECK_WORD,
  "Dsh",
  &DshMcl,
  (POBJ) &RshDummy,
  (POBJ) &RshDummy.dsh,
  &RshOffset }
};

Mms RshMms[] = {
  { MMS_CHECK_WORD, "Dsh", "destroy", RshDestroy },
  { MMS_CHECK_WORD, "Dsh", "display", RshDisplay },
  { MMS_CHECK_WORD, "Dsh", "erase",   RshErase },
  { MMS_CHECK_WORD, "Dsh", "rotate90",RshRotate90" }
};

Mcl RshMcl = {
    MCL_CHECK_WORD,
    "RectangleShape",
    sizeof(Class),
    sizeof(RectangleShape),
    RshMms,
    sizeof(RshMms)/sizeof(Mms),
    RshMsc,
    sizeof(RshMsc)/sizeof(Msc),
    MclCreateClass,
    ClsDestroy
};
```

Example 19 - Client Class RectangleShape

By creating a separate class for each display shape, we have factored display shapes into two type of code. That which is applicable to all display shapes regardless of type, and that which is specific to a particular display shape. This kind of factoring makes programs much more modular and understandable. It also makes adding new kinds of display shapes very easy.

In Example 20 we see how DisplayShape actually relies on the messages erase and display to implement the move function. Implementing the move function in this manner allows us to not have to rewrite it for each display shape!

```
Void    DshMove( PDSH pDsh, PPNT pPnt )
{
    Block   blk;

    BlkInit( &blk, NULL );
    ObjSendMessage( &pDsh->obj, MSG_DSH_ERASE, &blk );
    PntAddPnt( &pDsh->origin, pPnt );
    ObjSendMessage( &pDsh->obj, MSG_DSH_DISPLAY, &blk );
    BlkDeInit( &blk );
}
```

Example 20 - DisplayShape Creates a Framework for Moving Shapes

We have now seen how a class can be written which relies on client classes to perform some of the work. Such classes are called frameworks, or abstract superclasses. The benefit to this is that we can write programs which manipulate abstract objects of which there may be an unknown number of specific types. As new types are needed, they can easily be added into the system with almost no changes to the existing code.

Next, we show how to override messages which have the same name in 2 or more superclasses.

Overriding Messages and Multiple Inheritance

Multiple inheritance (conceptually) creates a slight problem when we have two or more superclasses which define the same message. The problem is that we need precision control when overriding messages so that we know exactly which superclass we are referring to when overriding a message. We also want to make sure that a message overridden by a subclass will trickle-down appropriately to the superclasses. Many OOPLs have a problem with multiple inheritance for this reason. C+O provides a great deal of control so that no ambiguity arises when overriding messages.

In Example 21, we see how the C+O class Edge overrides the destroy message from the three ListElements it inherits from (refer for a moment to the reference section on Edge, specifically the section on Class Implementation).

```
Msc  EdgMsc[] = {
{    MSC_CHECK_WORD,
     "GrfLel",
     &LelMcl,
     (POBJ) &EdgDummy,
     (POBJ) &EdgDummy.grfLel,
     &EdgGrfLelOffset
},
{    MSC_CHECK_WORD,
     "VtxInLel",
     &LelMcl,
     (POBJ) &EdgDummy,
     (POBJ) &EdgDummy.vtxInLel,
     &EdgVtxInLelOffset
},
{    MSC_CHECK_WORD,
     "VtxOutLel",
     &LelMcl,
     (POBJ) EdgDummy,
     (POBJ) &EdgDummy.vtxOutLel,
     &EdgVtxOutLelOffset
}
     };

Mms    EdgMms[] = {
  { MMS_CHECK_WORD, "GrfLel",      "destroy", EdgDestroy },
  { MMS_CHECK_WORD, "VtxInLel",    "destroy", EdgDestroy },
  { MMS_CHECK_WORD, "VtxOutLel", "destroy", EdgDestroy }
};
```

Example 21 - Edge Class Overrides the Same Message

Edge defines three superclasses, all of them ListElements. Each one is named differently: GrfLel, VtxInLel, and VtxOutLel. This allows us to select between superclasses by name without ambiguity.

Edge also defines three MetaMessages. Each MetaMessage can override a message from one and only one superclass. Therefore, the destroy message inherited by each ListElement must be overridden by a separate MetaMessage. Any one of the ListElement superclasses may send the destroy message to Edge so each one must be overridden explicitly.

Note that while we chose to override all three messages with the identical Edge function (EdgDestroy), we could have used a different Edge function for each one. This is very flexible and in practice quite useful.

But what happens when a client class of Edge overrides the destroy message? First of all, let's make clear that the client class of Edge cannot specify the particular destroy message to override. We don't even want the client class to care that there is more than one destroy message -- that is simply a function of the implementation of Edge.

The first destroy message which is found in the MetaMessage table can always be overridden by the client class. In our example then, a client of Edge could override the destroy function associated with GrfLel. So if the GrfLel ListElement were to send a destroy message, it will be properly routed to the client of Edge. But how about VtxInLel and VtxOutLel.

C+O uses the following algorithm to determine how to override the other destroy messages. If subsequent destroy messages listed for Edge have identical semantics (i.e. they call the same function as the first), then they will be overridden by a client class as well. Because in our example they do, destroy messages sent by VtxInLel and VtxOutLel will be overridden too. Note that because Edge has three identical messages it can choose one of three values for the destroy message and still call the same function.

The search for identical messages stops on the first non-match. Therefore, you should always group messages with the same name together.

What's In a Name?

One of the things that can help a programmer better understand his own code, or that of others, is if functions get named according to some conventions.

In C+O we have developed a naming system which strives to be consistent not only within a class, but across classes. The following are some names which get used consistently throughout C+O.

Init. Init functions are always used to initialize newly allocated memory. Each class should have an init function and should require that it be the first function to call prior to using any other functions for an object. The init function should initialize an object sufficiently so that it can be used by other functions. Init functions never return a value.

DeInit. This is the inverse of the Init function. DeInit is always capitalized as shown. It should be the last function called on an object just prior to deallocating its memory. Every class should provide a DeInit function and should require its usage prior to deallocation. DeInit should deallocate any memory
allocated to the object (other than the memory for the object). It should also ensure (to whatever degree possible) that it is not referenced by any other objects. Using an object which has been deinitialized is equivalent to using an object that has not been initialized and should result in an exception being triggered. DeInit functions never return a value.

Create. Create functions are used to create instance of other objects. For example, MclCreateCls creates objects of type Class. The return value from a Create function should always be a pointer to the newly created object. Create functions should always call the appropriate Init function before returning the object pointer. Most classes should create objects by calling ClsCreateObject.

Destroy. Destroy functions deinitialize and deallocate the memory for an object. The object should not be accessed after being destroyed. Destroy functions should first Clear the object or otherwise prepare it for deinitialization. It should then be deinitialized and finally the memory should be deallocated by calling ObjDestroy (assuming the object was allocated by ClsCreateObject). The Destroy function for a class always destroys an object of that type. Destroy functions never return a value.

Clear. Clear functions are provided by most every class and are used to set an object back to its state just after initialization.

Get. Get functions are used to retrieve values from an object. Typically the values are stored in the object but they may be calculated in some way as well. Get functions should always return something.

Is, Has, Can, etc. These functions are generally used to query an object about its state. For example, DllIsEmpty answers the question, "Is the list object empty?". The function name is always worded as a question to be answered. The return value is always a Bool.

Visit. Visit functions traverse an object which has some structure to it. Visit functions always take a Block object as a parameter, executing the Block for each object to be visited.

Client. Client functions are provided by classes which use advanced inheritance techniques to achieve reusability. Client Get functions always return objects which are a subclass of the object. Client Visit functions always visit the subclass objects of an object and so on.

Range. Range functions have no special function except to imply that two of the arguments to a function represent a beginning and ending element and all the elements in between.

Conclusion

This concludes our introduction to using C+O object-oriented programming facilities. For further examples, specifically on the use of individual classes in C+O, you may wish to look at the sample code provided with this package. C source code prefixed with TST demonstrates the use of many of the C+O classes.

This page is intentionally left blank

Class Description for *Block*

Structure Name: Block

Abbreviation: Blk

Class Type: Primitive Class

Introduction

The *Block* class is used to hold a function pointer and parameters to pass to that function. A block can be passed to other functions which have no knowledge of the contents of the block. The block can subsequently be asked to execute the function, passing the parameters stored with it.

Blocks are used as part of the messaging sending mechanism. They are also used by Visit functions (see *Tree* for extensive examples). You may wish to use them when creating your own Visit functions for a *Class*.

The term *Block* is a carryover from Smalltalk. Smalltalk uses them to implement **user-definable control structures**. In effect, that's exactly what Blocks allow you to do.

MetaClass, along with *Class*, *Object*, *Message*, *MetaSuperClass*, *MetaMessage*, and *Block* collectively define and implement the object-oriented properties of inheritance and dynamic binding (messaging) in C+O class libraries. Encapsulation, while supported by the above code, can be implemented by the programmer simply writing enough functions for a structure to obviate the need to access the structure members directly. These classes should be studied as a group to understand how C+O implements object-oriented C programs.

Block

Description

A *Block* can hold up to 16/32 bytes (depending on memory mode) of parameters to pass to a function. The object pointer which gets passed to *Block* on an execute call does not take up parameter space in the block.

A *Block* may not call functions where the funtion called pops the arguments off the stack. This should not be a problem unless the function being called is declared with the Pascal keyword. Later releases of C+O may remove this restriction.

Glossary and Special Terms

There is no special glossary for *Block*. Instead, refer to the tutorial section on object-oriented techniques in C+O.

Class Implementation *Block*

Instance Variables

The following is the data structure definition for a *Block*.

```
struct Block {      MediumInt     blkCheck;
                    PMTH          method;
                    USmallInt     index;
                    UMediumInt    parms[]; }
```

blkCheck	Uniquely identifies *Block* instances
method	Function to call
index	Keeps track of how many parameters have been pushed
parms	An array for storing parameters

Superclasses

The *Block* class is primitive and has no superclasses.

Messages and Responses

The *Block* class is primitive and has no messages.

Class Variables

The *Block* class has no class variables.

Function Listing In Alphabetic Order

Function Name	Description	Page	Scope	Macro
BlkClear	Clear instance	Blk - 2	Public	N
BlkDeInit	Deinitialize instance	Blk - 3	Public	N
BlkExecute	Execute method with parameters	Blk - 4	Public	N
BlkExecuteRetBool	Execute the method, return int	Blk - 5	Public	N
BlkExecuteRetDataPtr	Execute method, return mem pointer	Blk - 6	Public	N
BlkExecuteRetFuncPtr	Execute method, return function ptr	Blk - 7	Public	N
BlkExecuteRetInt	Execute method, return int	Blk - 8	Public	N
BlkHasMethod	Return True if instance has non-NULL method	Blk - 9	Public	N
BlkInit	Initialize instance	Blk - 10	Public	N
BlkPrint	Print contents of instance	Blk - 11	Public	N
BlkPushDataPtr	Save pointer parameter	Blk - 12	Public	N
BlkPushFuncPtr	Save function pointer parameter.	Blk - 13	Public	N
BlkPushLargeInt	Save LargeInt parameter	Blk - 14	Public	N
BlkPushMediumInt	Save MediumInt parameter	Blk - 15	Public	N
BlkSetMethod	Set the function to call	Blk - 16	Public	N

Function Listing By Scope

Public Functions

Function Name	Description	Page	Macro
BlkClear	Clear instance	Blk - 2	N
BlkDeInit	Deinitialize instance	Blk - 3	N
BlkExecute	Execute method with parameters	Blk - 4	N
BlkExecuteRetBool	Execute the method, return int	Blk - 5	N
BlkExecuteRetDataPtr	Execute method, return mem pointer	Blk - 6	N
BlkExecuteRetFuncPtr	Execute method, return function ptr	Blk - 7	N
BlkExecuteRetInt	Execute method, return int	Blk - 8	N
BlkHasMethod	Return True if instance has non-NULL method	Blk - 9	N
BlkInit	Initialize instance	Blk - 10	N
BlkPrint	Print contents of instance	Blk - 11	N
BlkPushDataPtr	Save pointer parameter	Blk - 12	N
BlkPushFuncPtr	Save function pointer parameter.	Blk - 13	N
BlkPushLargeInt	Save LargeInt parameter	Blk - 14	N
BlkPushMediumInt	Save MediumInt parameter	Blk - 15	N
BlkSetMethod	Set the function to call	Blk - 16	N

Private Functions

None			

Undocumented Functions

None			

Friend Functions

None			

Listing Of Functions by Category

Initialization Functions

Function Name	Description	Page	Scope	Macro
BlkClear	Clear instance	Blk - 2	Public	N
BlkDeInit	Deinitialize instance	Blk - 3	Public	N
BlkInit	Initialize instance	Blk - 10	Public	N

Execute Functions

Function Name	Description	Page	Scope	Macro
BlkExecute	Execute method with parameters	Blk - 4	Public	N
BlkExecuteRetBool	Execute the method, return int	Blk - 5	Public	N
BlkExecuteRetDataPtr	Execute method, return mem pointer	Blk - 6	Public	N
BlkExecuteRetFuncPtr	Execute method, return function ptr	Blk - 7	Public	N
BlkExecuteRetInt	Execute method, return int	Blk - 8	Public	N

Query Functions

Function Name	Description	Page	Scope	Macro
BlkHasMethod	Return True if instance has non-NULL method	Blk - 9	Public	N

Debug Functi ns

Function Name	Description	Page	Scope	Macro
BlkPrint	Print contents of instance	Blk - 11	Public	N

Push Functions

Function Name	Description	Page	Scope	Macro
BlkPushDataPtr	Save pointer parameter	Blk - 12	Public	N
BlkPushFuncPtr	Save function pointer parameter.	Blk - 13	Public	N
BlkPushLargeInt	Save LargeInt parameter	Blk - 14	Public	N
BlkPushMediumInt	Save MediumInt parameter	Blk - 15	Public	N
BlkSetMethod	Set the function to call	Blk - 16	Public	N

Listing Of Functions With Macros Available

None			

This page intentionally left blank

Class Description for *Class*

Structure Name: Class

Abbreviation: Cls

Class Type: Primitive Class

Introduction

The class *Class* is used to create instances of objects. This is probably all you will ever use *Class* for unless you are writing functions for the class *Object*.

Instances of *Class* are created by calling MclCreateClass and passing in an *MetaClass* instance which describes the *Class*.

MetaClass, along with *Class*, *Object*, *Message*, *MetaSuperClass*, *MetaMessage*, and *Block* collectively define and implement the object-oriented properties of inheritance and dynamic binding (messaging) in C+O class libraries. Encapsulation, while supported by the above code, can be implemented by the programmer simply writing enough functions for a structure to obviate the need to access the structure members directly. These classes should be studied as a group to understand how C+O implements object-oriented C programs.

Class

Description

A function of *Class* is to describe or model the organization of a particular C structure. Since C structures can be organized hierarchically, we must define the structure *Class*, which models this hierarchy, in a hierarchical, recursive fashion similar to how one might describe a tree structure. To borrow from *Tree* terminology, a class may have a parent class and children classes. Figure 1 shows an example of this hierarchy.

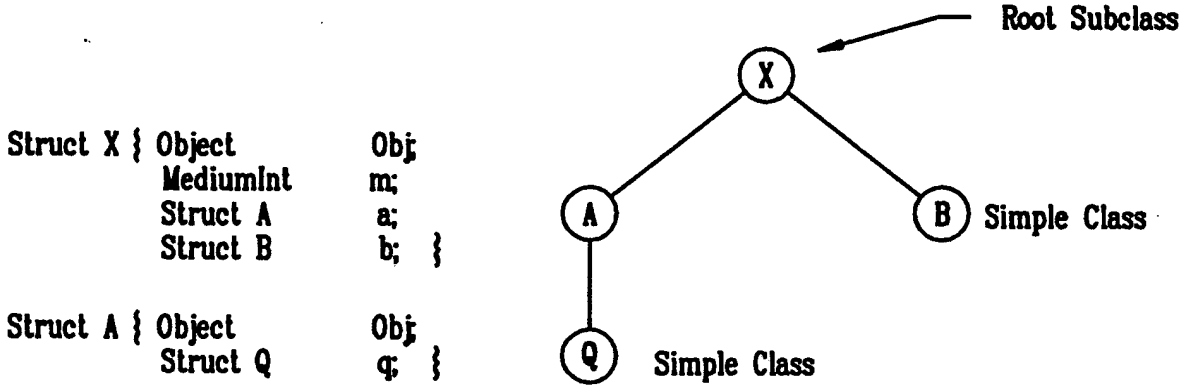


Figure 1

The *Class* structure does not try and model the complete description of a structure. It is only interested in modelling superclasses which are described with MetaClasses. Therefore, structure components such as ints, or even structures like *JulianTime* would not be described by a *Class*.

A *MetaClass* has one instantiation for each structure being defined. A *Class* on the other hand, has one instance *per use* of the structure. Therefore if a structure X is included in three other structures, there would be three instantiations of the *Class* X.

Glossary and Special Terms

The *Class* hierarchy has its own special terminlogy for a parent *Class* and children Classes. If X is an instance of *Class*, then the parent of X is called X's **subclass** and the children of X are called X's **superclasses**. X is said to **inherit** from its superclasses. If X has no subclass, it is the **root subclass**. If X has no superclasses, it is a **simple class**.

For other definitions, see the tutorial section on object-oriented techniques.

Class Implementation *Class*

Instance Variables

The following is the data structure definition for a *Class*.

```
struct Class {      MediumInt   clsCheck;
                    PMCL        mcl;
                    PCLS        subCls;
                    MediumInt   subOffset;
                    PCLS        mostSubCls;
                    MediumInt   mostSubOffset;
                    MediumInt   subIndex;
                    PCLS        *superClsArray;
                    PMSG        msgArray; }
```

clsCheck	Uniquely identifies *Class* instances.
mcl	The *MetaClass* which describes this class.
subCls	The subclass of this class (or NULL).
subOffset	The offset from the subclass.
mostSubCls	The root subclass.
mostSubOffset	The offset from the root subclass.
subIndex	The superclass index.
superClsArray	The array of superclass pointers.
msgArray	The array of messages for this class.

Superclasses

The class *Class* is primitive and has no superclasses.

Messages and Responses

The class *Class* is primitive and has no messages.

Class Variables

The class *Class* has no class variables.

Function Listing In Alphabetic Order

Function Name	Description	Page	Scope	Macro
ClsCreateMessages	Create the Msg instances	N/A	Undoc	N
ClsCreateObject	Create a new *Object* instance	Cls - 2	Public	N
ClsCreateSupers	Create the superclass instances	N/A	Undoc	N
ClsDeInit	Deinitialize the instance	Cls - 3	Public	N
ClsDestroy	Deallocate the instance	Cls - 4	Public	N
ClsDestroyMessages	Destroy the messages	N/A	Undoc	N
ClsDestroyObject	Deallocate the object instance	Cls - 5	Public	N
ClsDestroySuperClasses	Destroy the superclasses	N/A	Undoc	N
ClsFindMsg	Find message given selector name	Cls - 7	Public	N
ClsFindSelectorIndex	Find index of selector given name	Cls - 8	Public	N
ClsFindSuperClass	Find superclass given name	Cls - 10	Public	N
ClsGetMessageCount	Get number of messages	Cls - 11	Public	N
ClsGetMethodAndOffset	Return method and sub/super offset	Cls - 12	Public	N
ClsGetName	Return name of the instance	Cls - 14	Public	N
ClsGetNthMsg	Get pointer to nth *Message*	Cls - 15	Public	N
ClsGetNthSuperClass	Get pointer to Nth superclass	Cls - 16	Public	N
ClsGetObjectSize	Return size of an object instance	Cls - 17	Public	N
ClsGetOffsetForMsg	Return sub/super offset of object	Cls - 18	Public	N
ClsGetOffsetOfNthSuper	Return offset of nth superobject	Cls - 20	Public	N
ClsGetRootSubClass	Get the root subclass	Cls - 22	Public	N
ClsGetRootSubObjectOffset	Return offset of root subobject	Cls - 23	Public	N
ClsGetRootSubObjectSize	Return size of the root subobject	Cls - 24	Public	N
ClsGetSize	Return size of the instance	Cls - 25	Public	N
ClsGetSubClass	Return subclass	Cls - 26	Public	N
ClsGetSubObjectOffset	Return offset of a subobject	Cls - 27	Public	N
ClsGetSuperClassCount	Return number of superclasses	Cls - 28	Public	N
ClsGetSuperClassIndex	Return superclass index	Cls - 29	Public	N
ClsInit	Initialize the instance	Cls - 30	Public	N
ClsIsRoot	Is instance the root subclass?	Cls - 31	Public	N
ClsOffsetSupers	Offset superclasses from subclass.	N/A	Undoc	N
ClsPrint	Print the instance	Cls - 32	Public	N
ClsSendDestroy	Send destroy message to instance	Cls - 33	Public	N
ClsSendObjectMessage	Send object a message	Cls - 34	Public	N
ClsSendObjectMessageReturnInt	Send object message, returns MediumInt	Cls - 36	Public	N
ClsSendObjectMessageReturnPtr	Send object message, returns Void *	Cls - 38	Public	N
ClsSuperClassOf	Initialize sub/super relation	N/A	Undoc	N

Function Listing By Scope

Public Functions

Function Name	Description	Page	Macro
ClsCreateObject	Create a new *Object* instance	Cls - 2	N
ClsDeInit	Deinitialize the instance	Cls - 3	N
ClsDestroy	Deallocate the instance	Cls - 4	N
ClsDestroyObject	Deallocate the object instance	Cls - 5	N
ClsFindMsg	Find message given selector name	Cls - 7	N
ClsFindSelectorIndex	Find index of selector given name	Cls - 8	N
ClsFindSuperClass	Find superclass given name	Cls - 10	N
ClsGetMessageCount	Get number of messages	Cls - 11	N
ClsGetMethodAndOffset	Return method and sub/super offset	Cls - 12	N
ClsGetName	Return name of the instance	Cls - 14	N
ClsGetNthMsg	Get pointer to nth *Message*	Cls - 15	N
ClsGetNthSuperClass	Get pointer to Nth superclass	Cls - 16	N
ClsGetObjectSize	Return size of an object instance	Cls - 17	N
ClsGetOffsetForMsg	Return sub/super offset of object	Cls - 18	N
ClsGetOffsetOfNthSuper	Return offset of nth superobject	Cls - 20	N
ClsGetRootSubClass	Get the root subclass	Cls - 22	N
ClsGetRootSubObjectOffset	Return offset of root subobject	Cls - 23	N
ClsGetRootSubObjectSize	Return size of the root subobject	Cls - 24	N
ClsGetSize	Return size of the instance	Cls - 25	N
ClsGetSubClass	Return subclass	Cls - 26	N
ClsGetSubObjectOffset	Return offset of a subobject	Cls - 27	N
ClsGetSuperClassCount	Return number of superclasses	Cls - 28	N
ClsGetSuperClassIndex	Return superclass index	Cls - 29	N
ClsInit	Initialize the instance	Cls - 30	N
ClsIsRoot	Is instance the root subclass?	Cls - 31	N
ClsPrint	Print the instance	Cls - 32	N
ClsSendDestroy	Send destroy message to instance	Cls - 33	N
ClsSendObjectMessage	Send object a message	Cls - 34	N
ClsSendObjectMessageReturnInt	Send object message, returns MediumInt	Cls - 36	N
ClsSendObjectMessageReturnPtr	Send object message, returns Void *	Cls - 38	N

Private Functions

None			

Und cumented Functi ns

Function Name	Description	Page	Macro
ClsCreateMessages	Create the Msg instances	N/A	N
ClsCreateSupers	Create the superclass instances	N/A	N
ClsDestroyMessages	Destroy the messages	N/A	N
ClsDestroySuperClasses	Destroy the superclasses	N/A	N
ClsOffsetSupers	Offset superclasses from subclass.	N/A	N
ClsSuperClassOf	Initialize sub/super relation	N/A	N

Friend Functions

None			

Listing Of Functions by Category

Undocumented Functions

Function Name	Description	Page	Scope	Macro
ClsCreateMessages	Create the Msg instances	N/A	Undoc	N
ClsCreateSupers	Create the superclass instances	N/A	Undoc	N
ClsDestroyMessages	Destroy the messages	N/A	Undoc	N
ClsDestroySuperClasses	Destroy the superclasses	N/A	Undoc	N
ClsOffsetSupers	Offset superclasses from subclass.	N/A	Undoc	N
ClsSuperClassOf	Initialize sub/super relation	N/A	Undoc	N

Object Functions

Function Name	Description	Page	Scope	Macro
ClsCreateObject	Create a new *Object* instance	Cls - 2	Public	N
ClsDestroyObject	Deallocate the object instance	Cls - 5	Public	N
ClsSendObjectMessage	Send object a message	Cls - 34	Public	N
ClsSendObjectMessageReturnInt	Send object message, returns MediumInt	Cls - 36	Public	N
ClsSendObjectMessageReturnPtr	Send object message, returns Void *	Cls - 38	Public	N

Initialization Functions

Function Name	Description	Page	Scope	Macro
ClsDeInit	Deinitialize the instance	Cls - 3	Public	N
ClsDestroy	Deallocate the instance	Cls - 4	Public	N
ClsInit	Initialize the instance	Cls - 30	Public	N
ClsSendDestroy	Send destroy message to instance	Cls - 33	Public	N

Query Functions

Function Name	Description	Page	Scope	Macro
ClsFindMsg	Find message given selector name	Cls - 7	Public	N
ClsFindSelectorIndex	Find index of selector given name	Cls - 8	Public	N
ClsFindSuperClass	Find superclass given name	Cls - 10	Public	N
ClsGetMessageCount	Get number of messages	Cls - 11	Public	N
ClsGetMethodAndOffset	Return method and sub/super offset	Cls - 12	Public	N
ClsGetName	Return name of the instance	Cls - 14	Public	N
ClsGetNthMsg	Get pointer to nth *Message*	Cls - 15	Public	N
ClsGetNthSuperClass	Get pointer to Nth superclass	Cls - 16	Public	N
ClsGetObjectSize	Return size of an object instance	Cls - 17	Public	N
ClsGetOffsetForMsg	Return sub/super offset of object	Cls - 18	Public	N
ClsGetOffsetOfNthSuper	Return offset of nth superobject	Cls - 20	Public	N
ClsGetRootSubClass	Get the root subclass	Cls - 22	Public	N
ClsGetRootSubObjectOffset	Return offset of root subobject	Cls - 23	Public	N
ClsGetRootSubObjectSize	Return size of the root subobject	Cls - 24	Public	N
ClsGetSize	Return size of the instance	Cls - 25	Public	N
ClsGetSubClass	Return subclass	Cls - 26	Public	N
ClsGetSubObjectOffset	Return offset of a subobject	Cls - 27	Public	N
ClsGetSuperClassCount	Return number of superclasses	Cls - 28	Public	N
ClsGetSuperClassIndex	Return superclass index	Cls - 29	Public	N
ClsIsRoot	Is instance the root subclass?	Cls - 31	Public	N

Debug Functions

Function Name	Description	Page	Scope	Macro
ClsPrint	Print the instance	Cls - 32	Public	N

Listing Of Functions With Macros Available

None			

This page intentionally left blank

Class Description for *List*

Structure Name: List

Abbreviation: Dll

Class Type: Inheritable class

Introduction

The *List* class has a *friend* class called *ListElement*. The documentation for the *List* class should therefore be read in conjunction with the *ListElement* class.

When sequential mapping is used for ordered lists, achieved using an array data structure, operations such as insertion and deletion of arbitrary elements become expensive because of the amount of data movement. Linked representations of ordered lists are an alternative to sequential representations.

Description

Unlike a sequential representation where successive items of a list are located a fixed distance apart, in a linked representation these items may be placed anywhere in memory. Each element points to the next element in the list and the previous element in that list. These elements are stored as part of the class *ListElement*.

Besides the linked items a special node exists called the head node. The head node stores the first and last elements in the list. C+O stores the head node as part of the class *List*. *ListElements* also store a pointer to the *List* they belong to (if any).

It is customary to draw linked lists as an ordered sequence of nodes with links being represented by arrows. Figure 1 shows a sample doubly linked list with 4 nodes (*ListElements*) A, B, C and D. The head node (*List*) points to nodes A and D.

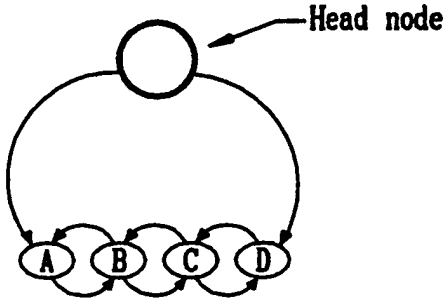


Figure 1

List

The *List* class is the data structure representation of the head node. The *ListElement* class is the data structure representation of a single element in a doubly linked list.

Glossary and Special Terms

Figure 2 shows the **first** and **last** element in a doubly linked list. Because each list element stores a pointer to the *List* class (**head** node) the first and last elements are also accessible from each element.

The **size** of a list is the number of elements in that list.

To get from the first element to the last element you would access **successor**, or **next** elements. To get from the last element to the first element you would access **predecessor** or **previous** elements.

The predecessor of the first element is NULL. The successor to the last element is NULL. All other elements will have non-NULL predecessors and successors. There exists only zero or one predecessor for a given element.

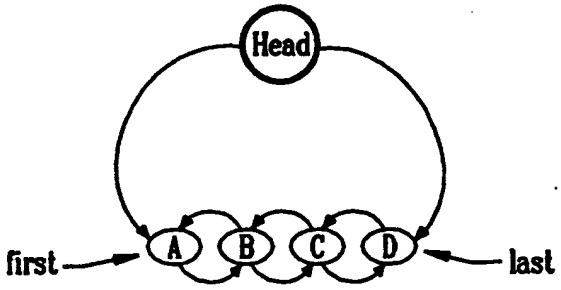


Figure 2

A **range** of list elements is defined as a **beginning** list element and an **ending** list element.

Examples of a valid range of list elements:

1) A through H, here the range is from first to last. 2) C through E, here the range spans elements C, D and E. 3) F through F, here the range spans one element, F.

Examples of an invalid range of list elements:

1) E through A, here the range is backwards. 2) A through I, here the range spans outside the list.

The **size** of a *List* is the number of elements it contains. Figure 3 shows a doubly linked list with size 8.

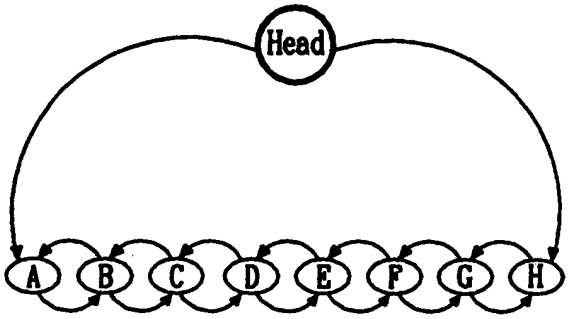


Figure 3

Class Implementation *List*

Instance Variables

The following is the data structure definition for a doubly linked list.

```
struct List {       Object      obj;
                    MediumInt   dllCheck;
                    PLEL        first;
                    PLEL        last;   }
```

obj	Gives *List* class object-oriented properties of inheritance and dynamic binding.
dllCheck	Uniquely identifies *List* instances.
first	First element in list.
last	Last element in list.

Messages and Responses

The implementation of the destroy message is handled by DllDestroy however subclasses should override this message. The destroy message is not used by any methods of Dll.

Selector	Method	Description
destroy	DllDestroy	Destroy the object

Class Variables

Type	Name	Description
Mcl	DllMcl	Metaclass description for *List*.

Function Listing In Alphabetic Order

Function Listing By Scope

Public Functions

Function Name	Description	Page	Macro
DllAppend	Append element to list	Dll - 2	N
DllAppendLast	Make element last	Dll - 4	N
DllClear	Clear the list	Dll - 7	N
DllCut	Cut one element from list	Dll - 8	N
DllCutChildren	Cut all elements from list	Dll - 10	N
DllCutRange	Cut element(s) from list	Dll - 11	N
DllDeInit	Deinitialize list object	Dll - 13	N
DllDestroy	Deinitialize list object and free space	Dll - 14	N
DllGetClient	Return client of list	Dll - 15	N
DllGetSize	Get size of list	Dll - 19	N
DllInit	Initialize list object	Dll - 20	N
DllInsert	Insert element in list	Dll - 21	N
DllInsertFirst	Make element first	Dll - 23	N
DllIsEmpty	Return True if list empty	Dll - 25	N
DllLelClientCount	Visit function: count elements conditionally	Dll - 26	N
DllLelClientFind	Visit search function: all elements	Dll - 28	N
DllLelClientFirst	Return client of first element	Dll - 30	N
DllLelClientGetNth	Return Nth client	Dll - 31	N
DllLelClientLast	Return client of last element	Dll - 33	N
DllLelClientVisitBwd	Visit function: all elements	Dll - 34	N
DllLelClientVisitFwd	Visit function: all elements	Dll - 36	N
DllPasteRangeAfter	Paste element(s) to list	Dll - 40	N
DllPasteRangeBefore	Paste element(s) in list	Dll - 42	N
DllPasteRangeFirst	Paste element(s) to be first in list	Dll - 44	N
DllPasteRangeLast	Paste element(s) to end of list	Dll - 46	N
DllSendDestroy	Send message for list destruction	Dll - 48	N

Private Functions

Function Name	Description	Page	Macro
DllAsObj	Return list as object	Dll - 6	N
DllGetFirst	Return first element	Dll - 16	N
DllGetLast	Return last element	Dll - 17	N
DllGetNth	Return Nth element	Dll - 18	N

Undocumented Functions

None			

Friend Functions

Function Name	Description	Page	Macro
DllNotifyCutRange	Cut elements	Dll - 38	N
DllNotifyPasteRange	Paste elements	Dll - 39	N

Listing Of Functions by Category

Edit Functions

Function Name	Description	Page	Scope	Macro
DllAppend	Append element to list	Dll - 2	Public	N
DllAppendLast	Make element last	Dll - 4	Public	Y
DllClear	Clear the list	Dll - 7	Public	N
DllCut	Cut one element from list	Dll - 8	Public	N
DllCutChildren	Cut all elements from list	Dll - 10	Public	N
DllCutRange	Cut element(s) from list	Dll - 11	Public	N
DllInsert	Insert element in list	Dll - 21	Public	N
DllInsertFirst	Make element first	Dll - 23	Public	Y
DllNotifyCutRange	Cut elements	Dll - 38	Friend	N
DllNotifyPasteRange	Paste elements	Dll - 39	Friend	N
DllPasteRangeAfter	Paste element(s) to list	Dll - 40	Public	N
DllPasteRangeBefore	Paste element(s) in list	Dll - 42	Public	N
DllPasteRangeFirst	Paste element(s) to be first in list	Dll - 44	Public	Y
DllPasteRangeLast	Paste element(s) to end of list	Dll - 46	Public	Y

Superclass Functions

Function Name	Description	Page	Scope	Macro
DllAsObj	Return list as object	Dll - 6	Private	Y

Initialization Functions

Function Name	Description	Page	Scope	Macro
DllDeInit	Deinitialize list object	Dll - 13	Public	N
DllDestroy	Deinitialize list object and free space	Dll - 14	Public	N
DllInit	Initialize list object	Dll - 20	Public	N
DllSendDestroy	Send message for list destruction	Dll - 48	Public	N

Query Functi ns

Function Name	Description	Page	Scope	Macro
DllGetClient	Return client of list	Dll - 15	Public	Y
DllGetFirst	Return first element	Dll - 16	Private	Y
DllGetLast	Return last element	Dll - 17	Private	Y
DllGetNth	Return Nth element	Dll - 18	Private	N
DllGetSize	Get size of list	Dll - 19	Public	N
DllIsEmpty	Return True if list empty	Dll - 25	Public	Y
DllLelClientCount	Visit function: count elements conditionally	Dll - 26	Public	N
DllLelClientFind	Visit search function: all elements	Dll - 28	Public	N
DllLelClientFirst	Return client of first element	Dll - 30	Public	Y
DllLelClientGetNth	Return Nth client	Dll - 31	Public	N
DllLelClientLast	Return client of last element	Dll - 33	Public	Y

Visit Functions

Function Name	Description	Page	Scope	Macro
DllLelClientVisitBwd	Visit function: all elements	Dll - 34	Public	N
DllLelClientVisitFwd	Visit function: all elements	Dll - 36	Public	N

Listing Of Functions With Macros Available

Function Name	Description	Page	Scope
DllAppendLast	Make element last	Dll - 4	Public
DllAsObj	Return list as object	Dll - 6	Private
DllGetClient	Return client of list	Dll - 15	Public
DllGetFirst	Return first element	Dll - 16	Private
DllGetLast	Return last element	Dll - 17	Private
DllInsertFirst	Make element first	Dll - 23	Public
DllIsEmpty	Return True if list empty	Dll - 25	Public
DllLelClientFirst	Return client of first element	Dll - 30	Public
DllLelClientLast	Return client of last element	Dll - 33	Public
DllPasteRangeFirst	Paste element(s) to be first in list	Dll - 44	Public
DllPasteRangeLast	Paste element(s) to end of list	Dll - 46	Public

Class Description for *DynamicArray*

Structure Name: DynamicArray

Abbreviation: Dpa

Class Type: Primitive

Introduction

The *DynamicArray* is a class for structuring data as an array of pointers such that elements can be quickly accessed by an index. The array in this case is dynamic, since it can grow and shrink at will. The next version of C+O will change the name *DynamicArray* to DynamicPointerArray. Please refer to the *DynamicArray* as Dpa when writing a program.

Applications of *DynamicArray* include graphical object arrays, text editor line display lists, stacks, queues, etc.

Because *DynamicArray* is a primitive class, it cannot inherit nor can it be inherited. A future version of C+O may change that.

Description

An array is a set of pairs, index and value. For each index which is defined, there is a value associated with that index. This is known as a **mapping**. The values in the *DynamicArray* class are 16 bit or 32 bit pointers (depending on memory model). Figure 1 shows an array of 6 elements with values A, B, C, D, E and F. The first element in the array A is at index zero.

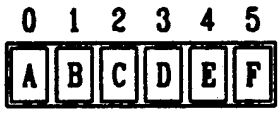


Figure 1

While elements of the array are not required to point to objects of the same class, *DynamicArray* is more valuable if they do. No loss of generality is encountered by doing this since the array element may point to a common abstract object type. Individual objects can then be of any subclass of the abstract type. (See section on classes for more details.)

While the array of elements may grow or shrink, the location of a *DynamicArray* object never changes. The array is grown or shrunk in multiples of a user specified value, thus changing the logical size of the array does not always result in a resizing of the physical array.

Glossary and Special Terms

The number of elements in an array is the **size** of the array. Figure 2 shows an array with 4 elements A, B, C and D. The size of the array is therefore 4.

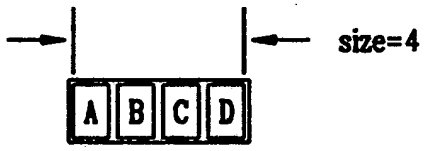


Figure 2

A **range** of array elements is defined as a **beginning element** and an **ending element**. Figure 3 shows an array with **size** 8. Possible valid ranges of elements are:

1) 0 through 7, here the range spans A, B, C, D, E, F, G, H. 2) 2 through 4, here the range spans elements C, D and E. 3) 5 through 5, here the range spans one element, F.

Possible invalid ranges of list elements are:

1) 4 through 0, here the range is backwards. 2) 0 through 8, here the range spans outside the array.

Figure 3

A **regi n** of array elements is defined as a beginning element and a **c unt** from the beginning element. Figure 4 shows an array with size 8. Possible valid regions of array elements are:

1) A with count 8, here the region is from first to last. 2) C with count 3, here the region spans elements C, D and E. 3) F with count 1, here the range spans one element, F.

Possible invalid regions of list elements are:

1) E with count 0, here the number of elements is less than 1. 2) C with count 9, here the count spans outside the array.

Figure 4

Class Implementation *DynamicArray*

Instance Variables

The following is the data structure definition for an array.

```
struct DynamicArray {      MediumInt     dpaCheck;
                           ClientPtr     *array;
                           MediumInt     size;
                           MediumInt     allocSize;
                           MediumInt     increment; }
```

dllCheck	Uniquely identifies *List* instances.
array	Holds the array values.
size	Logical size of array.
allocSize	Allocated size of array.
increment	Amount to grow the array by.

Superclasses

The *DynamicArray* class does not inherit from any other classes.

Class Variables

The *DynamicArray* class has no class instance variables.

Function Listing In Alphabetic Order

Function Name	Description	Page	Scope	Macro
DpaAppend	Append an element	Dpa - 2	Public	N
DpaClear	Clear dynamic array	Dpa - 4	Public	N
DpaCount	Visit function: count True returns	Dpa - 5	Public	N
DpaCountRange	Visit function: range with return checking	Dpa - 7	Public	N
DpaDeInit	Deinitialize the dynamic array object	Dpa - 9	Public	N
DpaDelete	Delete element(s)	Dpa - 10	Public	N
DpaDestroy	Deinitialize array object and free space	Dpa - 12	Public	N
DpaExpand	Paste Null element(s) into array	Dpa - 13	Public	N
DpaFind	Find index returning True	Dpa - 15	Public	N
DpaFindPtrBwd	Find index with matching pointer	Dpa - 17	Public	N
DpaFindPtrFwd	Find index with matching pointer	Dpa - 19	Public	N
DpaFindRangeBwd	Find index returning True for range	Dpa - 21	Public	N
DpaFindRangeFwd	Find index returning True for range	Dpa - 23	Public	N
DpaGetLast	Return last element in array	Dpa - 27	Public	N
DpaGetNth	Return Nth array element	Dpa - 25	Public	N
DpaGetSize	Return number of elements	Dpa - 29	Public	N
DpaInit	Initialize the edge object	Dpa - 31	Public	N
DpaNewArray	Create a new array	N/A	Undoc	N
DpaResize	Resize the array	N/A	Undoc	N
DpaSetNth	Set Nth element of array	Dpa - 33	Public	N
DpaSetRegionNull	Make region of elements Null	Dpa - 35	Public	N
DpaSetSize	Set array size to N elements	Dpa - 37	Public	N
DpaShiftDown	Shift down N elements in array	Dpa - 38	Public	N
DpaShiftUp	Shift up N elements in array	Dpa - 40	Public	N
DpaVisit	Visit function: all elements	Dpa - 42	Public	N
DpaVisitClient	Visit function: all elements	Dpa - 44	Public	N
DpaVisitRange	Visit function: range of elements	Dpa - 46	Public	N
DpaVisitRegion	Visit function: region of elements	Dpa - 48	Public	N
DpaVisitSelfAndSuccessors	Visit function: client and successors	Dpa - 50	Public	N

Function Listing By Scope

Public Functions

Function Name	Description	Page	Macro
DpaAppend	Append an element	Dpa - 2	N
DpaClear	Clear dynamic array	Dpa - 4	N
DpaCount	Visit function: count True returns	Dpa - 5	N
DpaCountRange	Visit function: range with return checking	Dpa - 7	N
DpaDeInit	Deinitialize the dynamic array object	Dpa - 9	N
DpaDelete	Delete element(s)	Dpa - 10	N
DpaDestroy	Deinitialize array object and free space	Dpa - 12	N
DpaExpand	Paste Null element(s) into array	Dpa - 13	N
DpaFind	Find index returning True	Dpa - 15	N
DpaFindPtrBwd	Find index with matching pointer	Dpa - 17	N
DpaFindPtrFwd	Find index with matching pointer	Dpa - 19	N
DpaFindRangeBwd	Find index returning True for range	Dpa - 21	N
DpaFindRangeFwd	Find index returning True for range	Dpa - 23	N
DpaGetLast	Return last element in array	Dpa - 27	N
DpaGetNth	Return Nth array element	Dpa - 25	N
DpaGetSize	Return number of elements	Dpa - 29	N
DpaInit	Initialize the edge object	Dpa - 31	N
DpaSetNth	Set Nth element of array	Dpa - 33	N
DpaSetRegionNull	Make region of elements Null	Dpa - 35	N
DpaSetSize	Set array size to N elements	Dpa - 37	N
DpaShiftDown	Shift down N elements in array	Dpa - 38	N
DpaShiftUp	Shift up N elements in array	Dpa - 40	N
DpaVisit	Visit function: all elements	Dpa - 42	N
DpaVisitClient	Visit function: all elements	Dpa - 44	N
DpaVisitRange	Visit function: range of elements	Dpa - 46	N
DpaVisitRegion	Visit function: region of elements	Dpa - 48	N
DpaVisitSelfAndSuccessors	Visit function: client and successors	Dpa - 50	N

Private Functions

None			

Undocumented Functions

Function Name	Description	Page	Macro
DpaNewArray	Create a new array	N/A	N
DpaResize	Resize the array	N/A	N

Friend Functions

None			

Listing Of Functions by Category

Edit Functions

Function Name	Description	Page	Scope	Macro
DpaAppend	Append an element	Dpa - 2	Public	N
DpaClear	Clear dynamic array	Dpa - 4	Public	N
DpaDelete	Delete element(s)	Dpa - 10	Public	N
Dé'Expand	Paste Null element(s) into array	Dpa - 13	Public	N
DpaNewArray	Create a new array	N/A	Undoc	N
DpaResize	Resize the array	N/A	Undoc	N
DpaSetNth	Set Nth element of array	Dpa - 33	Public	N
DpaSetRegionNull	Make region of elements Null	Dpa - 35	Public	N
DpaSetSize	Set array size to N elements	Dpa - 37	Public	N
DpaShiftDown	Shift down N elements in array	Dpa - 38	Public	N
DpaShiftUp	Shift up N elements in array	Dpa - 40	Public	N

Query Functions

Function Name	Description	Page	Scope	Macro
DpaCount	Visit function: count True returns	Dpa - 5	Public	N
DpaCountRange	Visit function: range with return checking	Dpa - 7	Public	N
DpaFind	Find index returning True	Dpa - 15	Public	N
DpaFindPtrBwd	Find index with matching pointer	Dpa - 17	Public	N
DpaFindPtrFwd	Find index with matching pointer	Dpa - 19	Public	N
DpaFindRangeBwd	Find index returning True for range	Dpa - 21	Public	N
DpaFindRangeFwd	Find index returning True for range	Dpa - 23	Public	N
DpaGetLast	Return last element in array	Dpa - 27	Public	N
DpaGetNth	Return Nth array element	Dpa - 25	Public	N
DpaGetSize	Return number of elements	Dpa - 29	Public	N

Initializati n Functions

Function Name	Description	Page	Scope	Macro
DpaDeInit	Deinitialize the dynamic array object	Dpa - 9	Public	N
DpaDestroy	Deinitialize array object and free space	Dpa - 12	Public	N
DpaInit	Initialize the edge object	Dpa - 31	Public	N

Visit Functions

Function Name	Description	Page	Scope	Macro
DpaVisit	Visit function: all elements	Dpa - 42	Public	N
DpaVisitClient	Visit function: all elements	Dpa - 44	Public	N
DpaVisitRange	Visit function: range of elements	Dpa - 46	Public	N
DpaVisitRegion	Visit function: region of elements	Dpa - 48	Public	N
DpaVisitSelfAndSuccessors	Visit function: client and successors	Dpa - 50	Public	N

Listing Of Functions With Macros Available

None			

Class Description for *Edge*

Structure Name: Edge

Abbreviation: Edg

Class Type: Inheritable class

Introduction

An *Edge* is a only a useful object in the context of a graph and vertices. Edges can be subclassed to represent for example, relationships between activities in a critical path application, interdependencies between cells in a spreadsheet, links in a hypertext document, etc.

The *Edge* class has two *friend* classes *Vertex* and *Graph*. The documentation for the *Edge* class should therefore be read in conjunction with the classes *Vertex* and *Graph*.

Description

An edge **connects** two vertices. An edge has a **tail** and a **head**. The vertex connected to the tail is the **incoming vertex** and the vertex connected to the head is the **outgoing vertex**. In an undirected graph, these distinctions are meaningless. In Figure 1 below the edge x has the outgoing vertex A and the incoming vertex B.

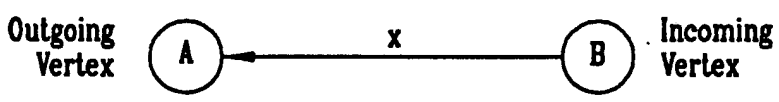


Figure 1

Edge

Glossary and Special Terms

The **predecessor vertex** of an edge is its incoming vertex and the **successor vertex** is its outgoing vertex. Similarly, the successor vertices of a *Vertex* A are the vertices at the head of the outgoing edges of A. The predecessor vertices of A are the vertices at the tail end of the incoming edges of A. See Figure 2.

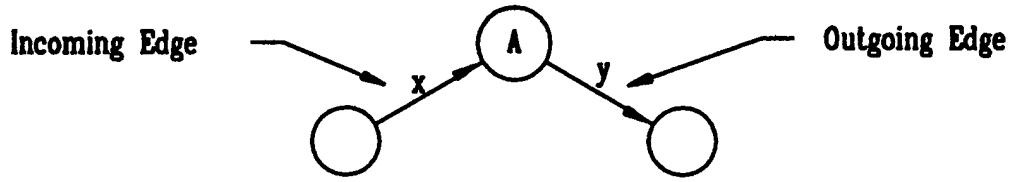


Figure 2

Class Implementation *Edge*

Instance Variables

The following is the data structure definition for a edge in a directed graph.

```
struct Edge {       Obj          obj;
                    MediumInt    edgCheck;
                    Lel          grfLel;
                    Lel          inLel;
                    Lel          outLel;   };
```

obj	Gives *Edge* class object-oriented properties of inheritance and dynamic binding.
edgCheck	Uniquely identifies *Edge* instances.
grfLel	*List* element in a graph.
inLel	*List* element of incoming edges for a vertex.
outLel	*List* element of outgoing edges for a vertex.

Superclasses

Type	Name	Description
ListElement	GrfLel	Connects edge to graph
ListElement	InLel	Connects edge to incoming vertex
ListElement	OutLel	Connects edge to outgoing vertex

Messages and Responses

Messages and Responses

Edge overrides the destroy message from all three of its *ListElement* superclasses. The implementation of the destroy message is handled by EdgDestroy. Subclasses of *Edge* should override the destroy message.

Selector	Method	Description
destroy	EdgDestroy	Destroy the object

Class Variables

Type	Name	Description
Mcl	EdgMcl	Metaclass decription for *Edge*.
MediumInt	EdgGrfLelOffset	Client offset of *Edge* from superclass GrfLel
MediumInt	EdgInLelOffset	Client offset of *Edge* from superclass InLel
MediumInt	EdgOutLelOffset	Client offset of *Edge* from superclass OutLel

Function Listing In Alphabetic Order

Function Name	Description	Page	Scope	Macro
EdgAsGrfLel	Return graph list element for edge	Edg - 2	Friend	Y
EdgAsInLel	Return incoming edge list element	Edg - 3	Friend	Y
EdgAsObj	Return edge as object	Edg - 4	Private	N
EdgAsOutLel	Return outgoing edge list element	Edg - 5	Friend	Y
EdgClear	Clear the edge	Edg - 6	Public	N
EdgCompareInVtx	Compare incoming vertex	Edg - 7	Public	N
EdgConnectToGrf	Connect edge to graph	Edg - 9	Public	Y
EdgConnectToVertices	Connect edge to vertices	Edg - 11	Public	N
EdgDeInit	Deinitialize the edge object	Edg - 13	Public	N
EdgDestroy	Deinitialize edge object and free space	Edg - 14	Public	N
EdgDisconnectFromGrf	Disconnect edge from graph	Edg - 15	Public	Y
EdgDisconnectFromVertices	Disconnect edge from vertices	Edg - 17	Public	N
EdgGetClient	Return client of edge	Edg - 19	Public	Y
EdgGetGrf	Return graph	Edg - 20	Friend	Y
EdgGetInVtx	Return incoming vertex	Edg - 21	Friend	Y
EdgGetNextIn	Return next incoming edge	Edg - 23	Friend	Y
EdgGetNextOut	Return next outgoing edge	Edg - 25	Friend	Y
EdgGetOutVtx	Return outgoing vertex	Edg - 27	Friend	Y
EdgGetVertices	Return vertices to edge	Edg - 29	Public	Y
EdgHasVertices	Does edge have any vertices	Edg - 31	Public	N
EdgInGrf	Is edge in graph	Edg - 32	Public	Y
EdgInit	Initialize the edge object	Edg - 33	Public	N
EdgSendDestroy	Send message for edge destruction	Edg - 34	Public	Y
EdgUpdateInVtx	Replace incoming vertex	Edg - 35	Public	N
EdgUpdateOutVtx	Replace outgoing vertex	Edg - 37	Public	N

Function Listing By Scope

Public Functions

Function Name	Description	Page	Macro
EdgClear	Clear the edge	Edg - 6	N
EdgCompareInVtx	Compare incoming vertex	Edg - 7	N
EdgConnectToGrf	Connect edge to graph	Edg - 9	N
EdgConnectToVertices	Connect edge to vertices	Edg - 11	N
EdgDeInit	Deinitialize the edge object	Edg - 13	N
EdgDestroy	Deinitialize edge object and free space	Edg - 14	N
EdgDisconnectFromGrf	Disconnect edge from graph	Edg - 15	N
EdgDisconnectFromVertices	Disconnect edge from vertices	Edg - 17	N
EdgGetClient	Return client of edge	Edg - 19	N
EdgGetVertices	Return vertices to edge	Edg - 29	N
EdgHasVertices	Does edge have any vertices	Edg - 31	N
EdgInGrf	Is edge in graph	Edg - 32	N
EdgInit	Initialize the edge object	Edg - 33	N
EdgSendDestroy	Send message for edge destruction	Edg - 34	N
EdgUpdateInVtx	Replace incoming vertex	Edg - 35	N
EdgUpdateOutVtx	Replace outgoing vertex	Edg - 37	N

Private Functions

Function Name	Description	Page	Macro
EdgAsObj	Return edge as object	Edg - 4	N

Undocumented Functions

None			

Friend Functions

Function Name	Description	Page	Macro
EdgAsGrfLel	Return graph list element for edge	Edg - 2	N
EdgAsInLel	Return incoming edge list element	Edg - 3	N
EdgAsOutLel	Return outgoing edge list element	Edg - 5	N
EdgGetGrf	Return graph	Edg - 20	N
EdgGetInVtx	Return incoming vertex	Edg - 21	N
EdgGetNextIn	Return next incoming edge	Edg - 23	N
EdgGetNextOut	Return next outgoing edge	Edg - 25	N
EdgGetOutVtx	Return outgoing vertex	Edg - 27	N

Listing Of Functions by Category

Superclass Functions

Function Name	Description	Page	Scope	Macro
EdgAsGrfLel	Return graph list element for edge	Edg - 2	Friend	Y
EdgAsInLel	Return incoming edge list element	Edg - 3	Friend	Y
EdgAsObj	Return edge as object	Edg - 4	Private	N
EdgAsOutLel	Return outgoing edge list element	Edg - 5	Friend	Y

Edit Functions

Function Name	Description	Page	Scope	Macro
EdgClear	Clear the edge	Edg - 6	Public	N
EdgConnectToGrf	Connect edge to graph	Edg - 9	Public	Y
EdgConnectToVertices	Connect edge to vertices	Edg - 11	Public	N
EdgDisconnectFromGrf	Disconnect edge from graph	Edg - 15	Public	Y
EdgDisconnectFromVertices	Disconnect edge from vertices	Edg - 17	Public	N
EdgUpdateInVtx	Replace incoming vertex	Edg - 35	Public	N
EdgUpdateOutVtx	Replace outgoing vertex	Edg - 37	Public	N

Query Functions

Function Name	Description	Page	Scope	Macro
EdgCompareInVtx	Compare incoming vertex	Edg - 7	Public	N
EdgGetClient	Return client of edge	Edg - 19	Public	Y
EdgGetGrf	Return graph	Edg - 20	Friend	Y
EdgGetInVtx	Return incoming vertex	Edg - 21	Friend	Y
EdgGetNextIn	Return next incoming edge	Edg - 23	Friend	Y
EdgGetNextOut	Return next outgoing edge	Edg - 25	Friend	Y
EdgGetOutVtx	Return outgoing vertex	Edg - 27	Friend	Y
EdgGetVertices	Return vertices to edge	Edg - 29	Public	Y
EdgHasVertices	Does edge have any vertices	Edg - 31	Public	N
EdgInGrf	Is edge in graph	Edg - 32	Public	Y

Initialization Functions

Function Name	Description	Page	Scope	Macro
EdgDeInit	Deinitialize the edge object	Edg - 13	Public	N
EdgDestroy	Deinitialize edge object and free space	Edg - 14	Public	N
EdgInit	Initialize the edge object	Edg - 33	Public	N
EdgSendDestroy	Send message for edge destruction	Edg - 34	Public	Y

Listing Of Functions With Macros Available

Function Name	Description	Page	Scope
EdgAsGrfLel	Return graph list element for edge	Edg - 2	Friend
EdgAsInLel	Return incoming edge list element	Edg - 3	Friend
EdgAsOutLel	Return outgoing edge list element	Edg - 5	Friend
EdgConnectToGrf	Connect edge to graph	Edg - 9	Public
EdgDisconnectFromGrf	Disconnect edge from graph	Edg - 15	Public
EdgGetClient	Return client of edge	Edg - 19	Public
EdgGetGrf	Return graph	Edg - 20	Friend
EdgGetInVtx	Return incoming vertex	Edg - 21	Friend
EdgGetNextIn	Return next incoming edge	Edg - 23	Friend
EdgGetNextOut	Return next outgoing edge	Edg - 25	Friend
EdgGetOutVtx	Return outgoing vertex	Edg - 27	Friend
EdgGetVertices	Return vertices to edge	Edg - 29	Public
EdgInGrf	Is edge in graph	Edg - 32	Public
EdgSendDestroy	Send message for edge destruction	Edg - 34	Public

This page intentionally left blank

Class Description for *Graph*

Structure Name: Graph

Abbreviation: Grf

Class Type: Inheritable class

Introduction

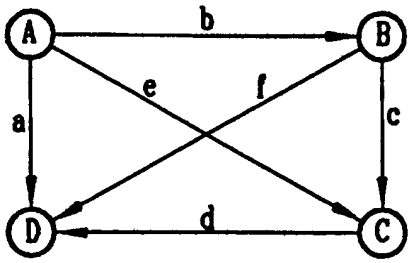
Graphs are used for a variety of applications. Some of these applications are: analysis of electrical circuits, finding shortest routes, analysis of project planning, identification of chemical compounds, statistical mechanics, genetics, cybernetics, CASE tools, etc. Of all mathematical structures, graphs are the most widely used.

The *Graph* class has two *friend* classes *Vertex* and *Edge*. The documentation for the *Graph* class should therefore be read in conjunction with the classes *Vertex* and *Edge*.

Description

A graph consists of a set of vertices and a set of edges, where each edge is a pair of distinct vertices.

Graph can deal with **directed graphs** or **digraphs**. In a directed graph each edge is represented by a directed pair of vertices, and the edges are drawn with an arrow from the tail to the head. A graph can have multiple occurances of the same edge (i.e. two edges with identical predecessor and successor vertices), it is up to the client of *Graph* to determine whether this should be allowed. In Figure 1 below the graph G contains the vertices A, B, C, D, and the edges a, b, c, d, e, f.



Graph G

Figure 1

Graph

Glossary and Special Terms

A vertex can have **incoming edges** and **outgoing edges**. In Figure 2 below the vertex A has an incoming edge x, and an outgoing edge y.

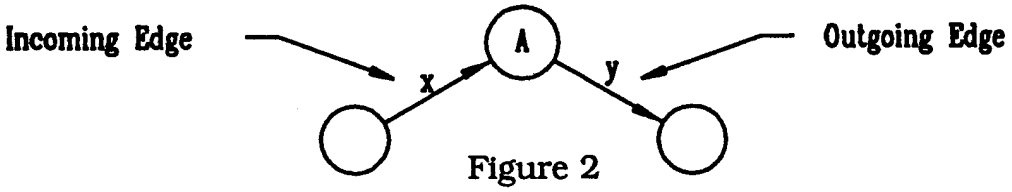


Figure 2

An edge **connects** two vertices. An edge has a **tail** and a **head**. The vertex connected to the tail is the **incoming vertex** and the vertex connected to the head is the **outgoing vertex**. In an undirected graph, these distinctions are meaningless. In Figure 3 below the edge x has the outgoing vertex A and the incoming vertex B.

The **predecessor vertex** of an edge is its incoming vertex and the **successor vertex** is its outgoing vertex. Similarly, the successor vertices of a *Vertex* A are the vertices at the head of the outgoing edges of A. The predecessor vertices of A are the vertices at the tail end of the incoming edges of A. See Figure 3.

Figure 3

A graph without cycles is **acyclic**. A **cycle** is a path of edges that return back to the starting point. In Figure 4 below the graph G, which has vertices A, B, C, and edges x, y, z, has a cycle and is, therefore, not acyclic.

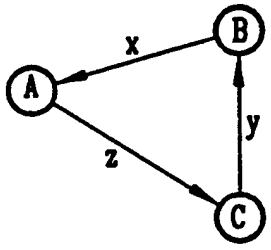


Figure 4

Class Implementation *Graph*

Instance Variables

The following is the data structure definition for a directed graph.

```
struct Graph {      Obj         obj;
                    MediumInt   grfCheck;
                    Dll         vtxList;
                    Dll         edgList;
                    Dpa         forwardSort;
                    Dpa         backwardSort; }
```

obj	Allows *Graph* to inherit and be inherited.
grfCheck	Uniquely identifies *Graph* instances.
vtxList	*List* of Vertices.
edgList	*List* of Edges.
forwardSort	Array storing vertices in forward topological order.
backwardSort	Array storing vertices in backward topological order.

Superclasses

Type	Name	Description
List	VtxDll	*List* of vertices
List	EdgDll	*List* of edges

Messages and Responses

Graph overrides the destroy message from both of its *List* superclasses. The implementation of the destroy message is handled by GrfDestroy. The subclass should override this message.

Selector	Method	Description
destroy	GrfDestroy	Destroy the object

Class Variables

Type	Name	Description
Mcl	GrfMcl	Metaclass decription for *Graph*.
MediumInt	GrfVtxDllOffset	Client offset of *Graph* from superclass VtxDll
MediumInt	GrfEdgDllOffset	Client offset of *Graph* from superclass EdgDll

Function Listing In Alphabetic Order

Function Name	Description	Page	Scope	Macro
GrfAnyCycles	Check graph for cycles	Grf - 2	Public	Y
GrfAsEdgDll	Return *List* of edges	Grf - 4	Friend	N
GrfAsObj	Return graph as object	Grf - 5	Private	N
GrfAsVtxDll	Return *List* of vertices	Grf - 6	Friend	N
GrfBasicTopologicalSort	Do topological sort of graph	N/A	Undoc	N
GrfClear	Clear the graph	Grf - 7	Public	N
GrfCountEdg	Count edges of graph	Grf - 8	Public	Y
GrfCountVtx	Count vertices of graph	Grf - 10	Public	Y
GrfDeInit	Deinitialize *Graph* object	Grf - 12	Public	N
GrfDestroy	Deinitialize *Graph* object and free space	Grf - 13	Public	N
GrfDoTopologicalSort	Do topological sort of graph	Grf - 14	Public	Y
GrfFindEdgClient	Visit search function: edges	Grf - 16	Public	N
GrfFindVtxClient	Visit search function: vertices	Grf - 18	Public	N
GrfGetClient	Return client of graph	Grf - 20	Public	N
GrfInit	Initialize *Graph* object	Grf - 21	Public	N
GrfSendDestroy	Send message for graph destruction	Grf - 22	Public	N
GrfVisitEdgClient	Visit function: edges	Grf - 23	Public	N
GrfVisitVtxClient	Visit function: vertices	Grf - 25	Public	N
GrfVisitVtxClientInTopOrderBwd	Visit function: backward topological order	Grf - 27	Public	N
GrfVisitVtxClientInTopOrderFwd	Visit function: forward topological order	Grf - 29	Public	N

Function Listing By Scope

Public Functions

Function Name	Description	Page	Macro
GrfAnyCycles	Check graph for cycles	Grf - 2	N
GrfClear	Clear the graph	Grf - 7	N
GrfCountEdg	Count edges of graph	Grf - 8	N
GrfCountVtx	Count vertices of graph	Grf - 10	N
GrfDeInit	Deinitialize *Graph* object	Grf - 12	N
GrfDestroy	Deinitialize *Graph* object and free space	Grf - 13	N
GrfDoTopologicalSort	Do topological sort of graph	Grf - 14	N
GrfFindEdgClient	Visit search function: edges	Grf - 16	N
GrfFindVtxClient	Visit search function: vertices	Grf - 18	N
GrfGetClient	Return client of graph	Grf - 20	N
GrfInit	Initialize *Graph* object	Grf - 21	N

Public Functi ns (cont)

Function Name	Description	Page	Macro
GrfSendDestroy	Send message for graph destruction	Grf - 22	N
GrfVisitEdgClient	Visit function: edges	Grf - 23	N
GrfVisitVtxClient	Visit function: vertices	Grf - 25	N
GrfVisitVtxClientInTopOrderBwd	Visit function: backward topological order	Grf - 27	N
GrfVisitVtxClientInTopOrderFwd	Visit function: forward topological order	Grf - 29	N

Private Functions

Function Name	Description	Page	Macro
GrfAsObj	Return graph as object	Grf - 5	N

Undocumented Functions

Function Name	Description	Page	Macro
GrfBasicTopologicalSort	Do topological sort of graph	N/A	N

Friend Functions

Function Name	Description	Page	Macro
GrfAsEdgDll	Return *List* of edges	Grf - 4	N
GrfAsVtxDll	Return *List* of vertices	Grf - 6	N

Listing Of Functions by Category

Query Functions

Function Name	Description	Page	Scope	Macro
GrfAnyCycles	Check graph for cycles	Grf - 2	Public	Y
GrfCountEdg	Count edges of graph	Grf - 8	Public	Y
GrfCountVtx	Count vertices of graph	Grf - 10	Public	Y
GrfFindEdgClient	Visit search function: edges	Grf - 16	Public	N
GrfFindVtxClient	Visit search function: vertices	Grf - 18	Public	N
GrfGetClient	Return client of graph	Grf - 20	Public	N

Superclass Functions

Function Name	Description	Page	Scope	Macro
GrfAsEdgDll	Return *List* of edges	Grf - 4	Friend	N
GrfAsObj	Return graph as object	Grf - 5	Private	N
GrfAsVtxDll	Return *List* of vertices	Grf - 6	Friend	N

Process Functions

Function Name	Description	Page	Scope	Macro
GrfBasicTopologicalSort	Do topological sort of graph	N/A	Undoc	N
GrfDoTopologicalSort	Do topological sort of graph	Grf - 14	Public	Y

Edit Functions

Function Name	Description	Page	Scope	Macro
GrfClear	Clear the graph	Grf - 7	Public	N

Initialization Functions

Function Name	Description	Page	Scope	Macro
GrfDeInit	Deinitialize *Graph* object	Grf - 12	Public	N
GrfDestroy	Deinitialize *Graph* object and free space	Grf - 13	Public	N
GrfInit	Initialize *Graph* object	Grf - 21	Public	N

Intialization Functions

Function Name	Description	Page	Scope	Macro
GrfSendDestroy	Send message for graph destruction	Grf - 22	Public	N

Visit Functions

Function Name	Description	Page	Scope	Macro
GrfVisitEdgClient	Visit function: edges	Grf - 23	Public	N
GrfVisitVtxClient	Visit function: vertices	Grf - 25	Public	N
GrfVisitVtxClientInTopOrderBwd	Visit function: backward topological order	Grf - 27	Public	N
GrfVisitVtxClientInTopOrderFwd	Visit function: forward topological order	Grf - 29	Public	N

Listing Of Functions With Macros Available

Function Name	Description	Page	Scope
GrfAnyCycles	Check graph for cycles	Grf - 2	Public
GrfCountEdg	Count edges of graph	Grf - 8	Public
GrfCountVtx	Count vertices of graph	Grf - 10	Public
GrfDoTopologicalSort	Do topological sort of graph	Grf - 14	Public

This page intentionally left blank

Class Description for *JulianTime*

Structure Name: JulianTime

Abbreviation: Jul

Class Type: Primitive

Introduction

The *JulianTime* class is used for modeling and manipulating calendar dates. It is best suited for applications where the calculations involving the arithmetical calculation of dates are more frequent than the formatting of dates for display.

The formatting functions provided by *JulianTime* are somewhat on the weak side. This class is really designed for date calculations. Future releases of this class will improve on its formatting capabilities.

Description

The *JulianTime* class provides functions for representing a specific day of a year as a julian date. A julian date is the number of days that have elapsed since noon GMT on January 1, 4713 BC. This is the julian day number. The consecutive numbering of days makes the system independent of the length of a month or year and is optimal for doing many date calculations.

Glossary and Special Terms

Julian Day Number:

The number of days that have elapsed since noon GMT on January 1, 4713 BC.

Example - Julian day on March 1st.

1940	=	2429690
1980	=	2444300
1990	=	2447952

Date Format:

JulianTime functions with formating arguments use an enum DateFormat as the argument. These format identifiers are as follows:

United States format	MM-DD-YYYY	has the enum DF_US.
European format	DD-MM-YYYY	has the enum DF_EUROPE.
Military format	YYYY-MM-DD	has the enum DF_MILITARY.

The algorithm implemented for the julian day conversion is valid for the Gregorian calendar. Valid years are therefore 1583 AD to 4713 AD.

Class Implementation *JulianTime*

Instance Variables

The structure definition for *JulianTime* is as follows:

struct *JulianTime* { ULargeInt time; }

time Julian day number.

Class Variables

The following class variables are used by the *JulianTime* class:

The class variable monthSize contains the size of each month of the year. The size of the second month (February) is preset to be a non leap year.

```
MediumInt monthSize[12]  = {31,28,31,30,31,30,31,31,30,31,30,31};
```

The class variable quarterSize contains the size of each quarter of the year.

```
MediumInt quarterSize[4] = {90,91,92,92};
```

The class variable weekMonthName contains the strings to be used to create the date string in the JulWeekString function.

```
Char weekMonthName[12][4] = { "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",
"Sep", "O 60"Nov", "Dec" };
```

The class variable MonthName contains the strings to be used to create the date string in the JulMonthString function.

```
Char MonthName[12][4] = { "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep",
"Oct", 60"Dec" };
```

The class variable QuarterName contains the strings to be used to create the date string in the JulQuarterString function.

```
Char QuarterName[4][8] = { "1st Qtr", "2nd Qtr", "3rd Qtr", "4th Qtr" };
```

Function Listing In Alphabetic Order

Function Name	Description	Page	Scope	Macro
JulAddDays	Add/subtract days to date	Jul - 2	Public	Y
JulAddDaysL	Add/subtract days to date (long)	Jul - 4	Public	Y
JulAddMonths	Add/subtract months to date	Jul - 6	Public	N
JulAddQuarters	Add/subtract quarters to date	Jul - 8	Public	N
JulAddYears	Add/subtract years to date	Jul - 10	Public	N
JulCalendarToJulian	Day, month, year to julian day	Jul - 12	Public	N
JulCopy	Copy julian day	Jul - 14	Public	Y
JulDateStrToJulian	Date string to julian day	Jul - 16	Public	N
JulDayOfWeek	Day number in week	Jul - 18	Public	Y
JulDayOfYear	Day number in year	Jul - 20	Public	N
JulDaysInMonth	Days in month	Jul - 22	Public	N
JulDaysInQuarter	Days in quarter	Jul - 24	Public	N
JulDaysInYear	Days in year	Jul - 26	Public	N
JulDiff	Days between two dates	Jul - 28	Public	Y
JulDiffL	Days between two dates (long)	Jul - 30	Public	Y
JulGetSystemJulianDay	System date as julian day	Jul - 32	Public	N
JulInit	Set to beginning of calendar January 1, 1583	Jul - 34	Public	Y
JulIsLeapYear	Is date in leap year	Jul - 36	Public	N
JulIsMaxValue	Is date maximum julian value	Jul - 38	Public	Y
JulMax	The maximum of two julian dates	Jul - 40	Public	Y
JulMin	The minimum of two julian dates	Jul - 42	Public	Y
JulMonthDayDiff	Days between date and a day/month	Jul - 44	Public	N
JulMonthString	Fill string with month and year	Jul - 46	Public	N
JulQuarterString	Fill string with quarter and year	Jul - 48	Public	N
JulSameDayMonth	Two dates same day and month	Jul - 50	Public	N
JulSetMaxDate	Set date to maximum value	Jul - 52	Public	N
JulToCalendar	Julian day to day, month, year	Jul - 54	Public	N
JulToDateStr	Fill date string of specified format	Jul - 56	Public	N
JulValidateDate	Validate date passed as string	Jul - 58	Public	N
JulWeekString	Fill string with day and month	Jul - 60	Public	N
JulYearString	Fill string with year	Jul - 62	Public	N

Function Listing By Scope

Public Functions

Function Name	Description	Page	Macro
JulAddDays	Add/subtract days to date	Jul - 2	N
JulAddDaysL	Add/subtract days to date (long)	Jul - 4	N
JulAddMonths	Add/subtract months to date	Jul - 6	N
JulAddQuarters	Add/subtract quarters to date	Jul - 8	N
JulAddYears	Add/subtract years to date	Jul - 10	N
JulCalendarToJulian	Day, month, year to julian day	Jul - 12	N
JulCopy	Copy julian day	Jul - 14	N
JulDateStrToJulian	Date string to julian day	Jul - 16	N
JulDayOfWeek	Day number in week	Jul - 18	N
JulDayOfYear	Day number in year	Jul - 20	N
JulDaysInMonth	Days in month	Jul - 22	N
JulDaysInQuarter	Days in quarter	Jul - 24	N
JulDaysInYear	Days in year	Jul - 26	N
JulDiff	Days between two dates	Jul - 28	N
JulDiffL	Days between two dates (long)	Jul - 30	N
JulGetSystemJulianDay	System date as julian day	Jul - 32	N
JulInit	Set to beginning of calendar January 1, 1583	Jul - 34	N
JulIsLeapYear	Is date in leap year	Jul - 36	N
JulIsMaxValue	Is date maximum julian value	Jul - 38	N
JulMax	The maximum of two julian dates	Jul - 40	N
JulMin	The minimum of two julian dates	Jul - 42	N
JulMonthDayDiff	Days between date and a day/month	Jul - 44	N
JulMonthString	Fill string with month and year	Jul - 46	N
JulQuarterString	Fill string with quarter and year	Jul - 48	N
JulSameDayMonth	Two dates same day and month	Jul - 50	N
JulSetMaxDate	Set date to maximum value	Jul - 52	N
JulToCalendar	Julian day to day, month, year	Jul - 54	N
JulToDateStr	Fill date string of specified format	Jul - 56	N
JulValidateDate	Validate date passed as string	Jul - 58	N
JulWeekString	Fill string with day and month	Jul - 60	N
JulYearString	Fill string with year	Jul - 62	N

Private Functions

None			

Undocumented Functions

None			

Friend Functions

None			

Listing Of Functions by Category

Arithmetical Functions

Function Name	Description	Page	Scope	Macro
JulAddDays	Add/subtract days to date	Jul - 2	Public	Y
JulAddDaysL	Add/subtract days to date (long)	Jul - 4	Public	Y
JulAddMonths	Add/subtract months to date	Jul - 6	Public	N
JulAddQuarters	Add/subtract quarters to date	Jul - 8	Public	N
JulAddYears	Add/subtract years to date	Jul - 10	Public	N
JulDiff	Days between two dates	Jul - 28	Public	Y
JulDiffL	Days between two dates (long)	Jul - 30	Public	Y
JulInit	Set to beginning of calendar January 1, 1583	Jul - 34	Public	Y
JulMax	The maximum of two julian dates	Jul - 40	Public	Y
JulMin	The minimum of two julian dates	Jul - 42	Public	Y
JulMonthDayDiff	Days between date and a day/month	Jul - 44	Public	N

Conversion Functions

Function Name	Description	Page	Scope	Macro
JulCalendarToJulian	Day, month, year to julian day	Jul - 12	Public	N
JulDateStrToJulian	Date string to julian day	Jul - 16	Public	N
JulGetSystemJulianDay	System date as julian day	Jul - 32	Public	N
JulToCalendar	Julian day to day, month, year	Jul - 54	Public	N
JulValidateDate	Validate date passed as string	Jul - 58	Public	N

Miscellaneous Functions

Function Name	Description	Page	Scope	Macro
JulCopy	Copy julian day	Jul - 14	Public	Y
JulSetMaxDate	Set date to maximum value	Jul - 52	Public	N

Query Functions

Function Name	Description	Page	Scope	Macro
JulDayOfWeek	Day number in week	Jul - 18	Public	Y
JulDayOfYear	Day number in year	Jul - 20	Public	N
JulDaysInMonth	Days in month	Jul - 22	Public	N
JulDaysInQuarter	Days in quarter	Jul - 24	Public	N
JulDaysInYear	Days in year	Jul - 26	Public	N
JulIsLeapYear	Is date in leap year	Jul - 36	Public	N
JulIsMaxValue	Is date maximum julian value	Jul - 38	Public	Y
JulSameDayMonth	Two dates same day and month	Jul - 50	Public	N

Formatting Functions

Function Name	Description	Page	Scope	Macro
JulMonthString	Fill string with month and year	Jul - 46	Public	N
JulQuarterString	Fill string with quarter and year	Jul - 48	Public	N
JulToDateStr	Fill date string of specified format	Jul - 56	Public	N
JulWeekString	Fill string with day and month	Jul - 60	Public	N
JulYearString	Fill string with year	Jul - 62	Public	N

Listing Of Functions With Macros Available

Function Name	Description	Page	Scope
JulAddDays	Add/subtract days to date	Jul - 2	Public
JulAddDaysL	Add/subtract days to date (long)	Jul - 4	Public
JulCopy	Copy julian day	Jul - 14	Public
JulDayOfWeek	Day number in week	Jul - 18	Public
JulDiff	Days between two dates	Jul - 28	Public
JulDiffL	Days between two dates (long)	Jul - 30	Public
JulInit	Set to beginning of calendar January 1, 1583	Jul - 34	Public
JulIsMaxValue	Is date maximum julian value	Jul - 38	Public
JulMax	The maximum of two julian dates	Jul - 40	Public
JulMin	The minimum of two julian dates	Jul - 42	Public

This page intentionally left blank

Class Description for *ListElement*

Structure Name: ListElement

Abbreviation: Lel

Class Type: Inheritable class

Introduction

The *ListElement* class has a *friend* class *List*. The documentation for the *ListElement* class should therefore be read in conjunction with the *List* class.

When sequential mapping is used for ordered lists, achieved using an array data structure, operations such as insertion and deletion of arbitrary elements become expensive because of the amount of data movement. Linked representations of ordered lists are an alternative to sequential representations.

Description

Unlike a sequential representation where successive items of a list are located a fixed distance apart, in a linked representation these items may be placed anywhere in memory. Each element points to the next element in the list and the previous element in that list. These elements are stored as part of the class *ListElement*.

Besides the linked items a special node exists called the head node. The head node stores the first and last elements in the list. C+O stores the head node as part of the class *List*. *ListElements* also store a pointer to the *List* they belong to (if any).

It is customary to draw linked lists as an ordered sequence of nodes with links being represented by arrows. Figure 1 shows a sample doubly linked list with 4 nodes (*ListElements*) A, B, C and D. The head node (*List*) points to nodes A and D.

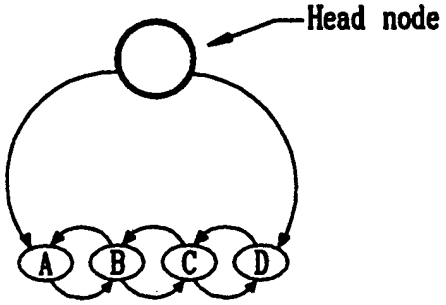


Figure 1

ListElement

The *List* class is the data structure representation of the head node. The *ListElement* class is the data structure representation of a single element in a doubly linked list.

Glossary and Special Terms

Figure 2 shows the **first** and **last** element in a doubly linked list. Because each list element stores a pointer to the *List* class (**head** node) the first and last elements are also accessible from each element.

To get from the first element to the last element you would access **successor**, or **next** elements. To get from the last element to the first element you would access **predecessor** or **previous** elements.

The predecessor of the first element is NULL. The successor to the last element is NULL. All other elements will have non-NULL predecessors and successors. There exists only zero or one predecessor for a given element.

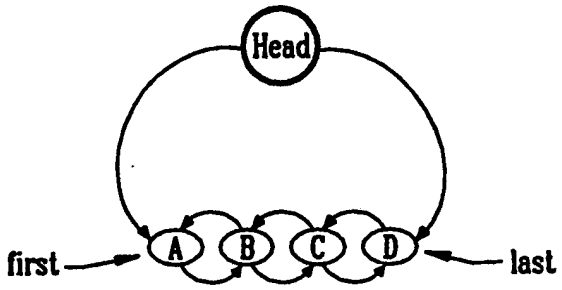


Figure 2

A **range** of list elements is defined as a **beginning** list element and an **ending** list element. Figure 3 shows a doubly linked list with size 8.

Examples of a valid range of list elements:

1) A through H, here the range is from first to last. 2) C through E, here the range spans elements C, D and E. 3) F through F, here the range spans one element, F.

Examples of an invalid range of list elements:

1) E through A, here the range is backwards. 2) A through I, here the range spans outside the list.

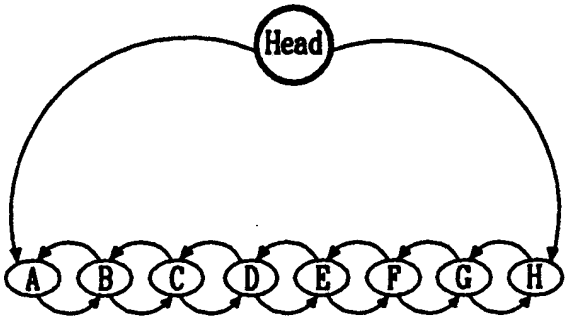


Figure 3

Class Implementation *ListElement*

Instance Variables

The following is the data structure definition for a list element.

```
struct ListElement {    Obj          obj;
                        MediumInt    lelCheck;
                        PDLL         pDll;
                        PLEL         prev;
                        PLEL         next;  }
```

obj	Gives *ListElement* class properties of inheritance and dynamic binding.
lelCheck	Uniquely identifies *ListElement* instances.
pDll	Pointer to *List* that the *ListElement* belongs to.
prev	Previous *ListElement* in the list.
next	Next *ListElement* in the list.

ListElement

Messages and Responses

The implementation of the destroy message is handled by LelDestroy but should be overridden by the subclass. The destroy message is not used by any Lel methods.

Selector	Method	Description
destroy	LelDestroy	Destroy the object

Class Variables

Type	Name	Description
Mcl	LelMcl	Metaclass description for *ListElement*.

Function Listing In Alphabetic Order

Function Listing By Scope

Public Functions

Function Name	Description	Page	Macro
LelClientCount	Return count for client and successors	Lel - 3	N
LelClientDll	Return client of list	Lel - 5	N
LelClientFindRange	Visit search function: range	Lel - 6	N
LelClientNext	Return client of next element	Lel - 8	N
LelClientPrev	Return client of previous element	Lel - 9	N
LelClientVisitBwd	Visit function: client and predecessors	Lel - 10	N
LelClientVisitFwd	Visit function: client and successors	Lel - 12	N
LelClientVisitPredecessors	Visit function: predecessors	Lel - 14	N
LelClientVisitRange	Visit function: range	Lel - 16	N
LelClientVisitSuccessors	Visit function: successors	Lel - 18	N
LelCountRange	Count elements	Lel - 20	N
LelCut	Cut element from list	Lel - 22	N
LelCutRange	Cut element(s) from list	Lel - 24	N
LelDeInit	Deinitialize list element object	Lel - 26	N
LelDestroy	Deinitialize list element object and free space	Lel - 27	N
LelGetClient	Return client	Lel - 28	N
LelInit	Is element in list	Lel - 33	N
LelPasteRangeAfter	Append elements(s) to list	Lel - 34	N
LelPasteRangeBefore	Insert element(s) to list	Lel - 36	N
LelSendDestroy	Send message for list element destruction	Lel - 38	N
LelTest	Test for valid list element	Lel - 39	N

Private Functions

Function Name	Description	Page	Macro
LelAsObj	Return element as object	Lel - 2	N
LelVisitRange	Visit function for range	Lel - 40	N

Undocumented Functions

Function Name	Description	Page	Macro
LelCutRangeFromList	Cut element(s) from list	N/A	N
LelPasteRangeToList	Insert element(s) to list	N/A	N

Friend Functions

Function Name	Description	Page	Macro
LelGetDll	Return list object	Lel - 29	N
LelGetNext	Return next element	Lel - 30	N
LelGetNthSuccessor	Return Nth element	Lel - 31	N
LelGetPrev	Return previous element	Lel - 32	N

Listing Of Functions by Category

Superclass Functions

Function Name	Description	Page	Scope	Macro
LelAsObj	Return element as object	Lel - 2	Private	Y

Query Functions

Function Name	Description	Page	Scope	Macro
LelClientCount	Return count for client and successors	Lel - 3	Public	N
LelClientDll	Return client of list	Lel - 5	Public	Y
LelClientFindRange	Visit search function: range	Lel - 6	Public	N
LelClientNext	Return client of next element	Lel - 8	Public	Y
LelClientPrev	Return client of previous element	Lel - 9	Public	Y
LelCountRange	Count elements	Lel - 20	Public	N
LelGetClient	Return client	Lel - 28	Public	Y
LelGetDll	Return list object	Lel - 29	Friend	Y
LelGetNext	Return next element	Lel - 30	Friend	Y
LelGetNthSuccessor	Return Nth element	Lel - 31	Friend	N
LelGetPrev	Return previous element	Lel - 32	Friend	Y
LelInit	Is element in list	Lel - 33	Public	N

Visit Functions

Function Name	Description	Page	Scope	Macro
LelClientVisitBwd	Visit function: client and predecessors	Lel - 10	Public	N
LelClientVisitFwd	Visit function: client and successors	Lel - 12	Public	N
LelClientVisitPredecessors	Visit function: predecessors	Lel - 14	Public	N
LelClientVisitRange	Visit function: range	Lel - 16	Public	N
LelClientVisitSuccessors	Visit function: successors	Lel - 18	Public	N
LelVisitRange	Visit function for range	Lel - 40	Private	N

Edit Functions

Function Name	Description	Page	Scope	Macro
LelCut	Cut element from list	Lel - 22	Public	N
LelCutRange	Cut element(s) from list	Lel - 24	Public	N
LelCutRangeFromList	Cut element(s) from list	N/A	Undoc	N
LelPasteRangeAfter	Append elements(s) to list	Lel - 34	Public	N
LelPasteRangeBefore	Insert element(s) to list	Lel - 36	Public	N
LelPasteRangeToList	Insert element(s) to list	N/A	Undoc	N

Initialization Functions

Function Name	Description	Page	Scope	Macro
LelDeInit	Deinitialize list element object	Lel - 26	Public	N
LelDestroy	Deinitialize list element object and free space	Lel - 27	Public	N
LelSendDestroy	Send message for list element destruction	Lel - 38	Public	N

Debug Functions

Function Name	Description	Page	Scope	Macro
LelTest	Test for valid list element	Lel - 39	Public	N

Listing Of Functions With Macros Available

Function Name	Description	Page	Scope
LelAsObj	Return element as object	Lel - 2	Private
LelClientDll	Return client of list	Lel - 5	Public
LelClientNext	Return client of next element	Lel - 8	Public
LelClientPrev	Return client of previous element	Lel - 9	Public
LelGetClient	Return client	Lel - 28	Public
LelGetDll	Return list object	Lel - 29	Friend
LelGetNext	Return next element	Lel - 30	Friend
LelGetPrev	Return previous element	Lel - 32	Friend

Class Description for *MetaClass*

Structure Name: MetaClass

Abbreviation: Mcl

Class Type: Primitive Class

Introduction

MetaClasses are used to create objects of type *Class*. By analogy, a *MetaClass* is to a *Class*, what a *Class* is to an *Object*. While ultimately it is Objects (instantiations of some structure or type) which we are interested in, we start by defining an instance of *MetaClass*.

Whenever you define a structure (class) which you want to have the object-oriented properties of inheritance or dynamic binding, you must do three things and in the following order.

First, the structure must contain, as the first structure member, a structure of type *Object*. Objects allow you to send messages (dynamic binding) and access subclasses (a class which inherits from your structure).

Second, you must define and initialize an instance of the structure *MetaClass* which describes the *Class*, its messages, and its superclass (classes which your class inherits from). Normally you will initialize the *MetaClass* instance at compile time.

Third, your program must call MclSendCreateClass, passing the *MetaClass* instance you defined in Step 2. The return value is (a pointer to) an instance of *Class*.

Now your program can create objects (memory allocated for the structure) of the *Class* you defined by calling ClsCreateObject and passing the pointer returned by MclSendCreateClass. When an object is no longer needed, you call ObjDestroy which deallocates the memory for that object.

MetaClass

MetaClass, along with *Class*, *Object*, *Message*, *MetaSuperClass*, *MetaMessage*, and *Block* collectively define and implement the object-oriented properties of inheritance and dynamic binding (messaging) in C+O class libraries. Encapsulation, while supported by the above code, can be implemented by the programmer simply writing enough functions for a structure to obviate the need to access the structure members directly. These classes should be studied as a group to understand how C+O implements object-oriented C programs.

For further information, read the tutorial section and also study the class header files dllcls.h, lelcls.h, and trecls.h. These headers define instances of *MetaClass*, *MetaSuperClass* and *MetaMessage*.

Description

An instance of *MetaClass* references an array of MetaSuperClasses and an array of MetaMessages. In its simplest form, an instance of *MetaClass* need only describe the name of the *Class* and the size of an object of that *Class*. Please refer to the sections on *MetaMessage* and MetaSuperclass for more information on defining instances of those classes.

A *MetaClass* is used to define all the constant information about a structure; it defines its size, the messages it responds to, and its superclasses.

The reader should famialiarize himself with the structure definition of *MetaClass* since you will be initializing instances of *MetaMessage* statically.

Glossary and Special Terms

There is no special glossary for *MetaClass*. Instead, refer to the tutorial section on object-oriented techniques in C+O.

Class Implementation *MetaClass*

Instance Variables

The following is the data structure definition for a directed graph.

```
struct MetaClass {      MediumInt    mclCheck;
                        PSTR         name;
                        MediumInt    clsSize;
                        MediumInt    objSize;
                        PMMS         mmsArray;
                        MediumInt    mmsSize;
                        PMSC         mscArray;
                        MediumInt    mscSize;
                        PCLS         (*createFunc)(PMCL);
                        Void         (*destroyFunc)(PCLS); }
```

mclCheck	Uniquely identifies *MetaClass* instances.
name	Name of the structure being modeled.
clsSize	Size of the *Class* structure or extended *Class* structure
objSize	Size of the structure being modeled.
mmsArray	An array of MetaMessages.
mmsSize	Number of MetaMessages in mmsArray.
mscArray	An array of MetaSuperClasses.
mscSize	Number of MetaSuperClasses.
createFunc	A function which will create a *Class* instance.
destroyFunc	A function which will destroy a *Class* instance.

Superclasses

MetaClass has no superclasses.

Messages and Responses

MetaClass does not have any messages since it is a primitive class.

Class Variables

MetaClass has no class variables.

MetaClass

Function Listing In Alphabetic Order

Function Name	Description	Page	Scope	Macro
MclCreateClass	Create a class instance	Mcl - 2	Private	N
MclDestroyClass	Destroy a class instance	Mcl - 3	Friend	N
MclFindSelector	Find a selector index given its name	Mcl - 5	Public	N
MclFindSuperClass	Find the superclass index given its name	Mcl - 6	Public	N
MclGetClassName	Get the class name	Mcl - 7	Public	N
MclGetClassSize	Return the size of a class instance	Mcl - 8	Public	N
MclGetMessageCount	Return the number of messages	Mcl - 9	Public	N
MclGetNthMms	Return the Nth *MetaMessage*.	Mcl - 10	Public	N
MclGetNthOffset	Return the Nth SuperClass offset	Mcl - 11	Public	N
MclGetNthSuper	Return the *MetaClass*	Mcl - 12	Public	N
MclGetSuperClassCount	Return the number of superclasses	Mcl - 13	Public	N
MclPrint	Print the *MetaClass* instance	Mcl - 14	Public	N
MclSendCreateClass	Return a new instance of the *MetaClass*	Mcl - 15	Public	N
MclSendDestroyClass	Deallocate an instance of a *Class*	Mcl - 16	Public	N
MclValidate	Validate that the instance is valid	Mcl - 17	Public	N
MclValidateMessages	Validate the MetaMessages	N/A	Undoc	N
MclValidateSuperClasses	Validate the MetaSuperClasses	N/A	Undoc	N

Function Listing By Scope

Public Functions

Function Name	Description	Page	Macro
MclFindSelector	Find a selector index given its name	Mcl - 5	N
MclFindSuperClass	Find the superclass index given its name	Mcl - 6	N
MclGetClassName	Get the class name	Mcl - 7	N
MclGetClassSize	Return the size of a class instance	Mcl - 8	N
MclGetMessageCount	Return the number of messages	Mcl - 9	N
MclGetNthMms	Return the Nth *MetaMessage*	Mcl - 10	N
MclGetNthOffset	Return the Nth SuperClass offset	Mcl - 11	N
MclGetNthSuper	Return the *MetaClass*	Mcl - 12	N
MclGetSuperClassCount	Return the number of superclasses	Mcl - 13	N
MclPrint	Print the *MetaClass* instance	Mcl - 14	N
MclSendCreateClass	Return a new instance of the *MetaClass*	Mcl - 15	N
MclSendDestroyClass	Deallocate an instance of a *Class*	Mcl - 16	N
MclValidate	Validate that the instance is valid	Mcl - 17	N

Private Functions

Function Name	Description	Page	Macro
MclCreateClass	Create a class instance	Mcl - 2	N

Undocumented Functions

Function Name	Description	Page	Macro
MclValidateMessages	Validate the MetaMessages	N/A	N
MclValidateSuperClasses	Validate the MetaSuperClasses	N/A	N

Friend Functions

Function Name	Description	Page	Macro
MclDestroyClass	Destroy a class instance	Mcl - 3	N

Listing Of Functions by Category

Class Functions

Function Name	Description	Page	Scope	Macro
MclCreateClass	Create a class instance	Mcl - 2	Private	N
MclDestroyClass	Destroy a class instance	Mcl - 3	Friend	N
MclSendCreateClass	Return a new instance of the *MetaClass*	Mcl - 15	Public	N
MclSendDestroyClass	Deallocate an instance of a *Class*	Mcl - 16	Public	N

Query Functions

Function Name	Description	Page	Scope	Macro
MclFindSelector	Find a selector index given its name	Mcl - 5	Public	N
MclFindSuperClass	Find the superclass index given its name	Mcl - 6	Public	N
MclGetClassName	Get the class name	Mcl - 7	Public	N
MclGetClassSize	Return the size of a class instance	Mcl - 8	Public	N
MclGetMessageCount	Return the number of messages	Mcl - 9	Public	N
MclGetNthMms	Return the Nth *MetaMessage*	Mcl - 10	Public	N
MclGetNthOffset	Return the Nth SuperClass offset	Mcl - 11	Public	N
MclGetNthSuper	Return the *MetaClass*	Mcl - 12	Public	N
MclGetSuperClassCount	Return the number of superclasses	Mcl - 13	Public	N

Debug Functions

Function Name	Description	Page	Scope	Macro
MclPrint	Print the *MetaClass* instance	Mcl - 14	Public	N
MclValidate	Validate that the instance is valid	Mcl - 17	Public	N
MclValidateMessages	Validate the MetaMessages	N/A	Undoc	N
MclValidateSuperClasses	Validate the MetaSuperClasses	N/A	Undoc	N

Listing Of Functions With Macros Available

None			

Class Description for *Memory*

Structure Name: Memory

Abbreviation: Mem

Class Type: Primitive Class

Introduction

The *Memory* class is used to manipulate arrays of bytes. Each element can be an arbitrary number of bytes wide. It is used by the *DynamicArray* class to implement many low level functions.

Description

The *Memory* class is an abstract representation of raw, unstructured sequential bytes of memory. It is the most primitive class in C+O. Many other classes use Mem for allocation and deallocation of memory.

Glossary and Special Terms

There is no special glossary for this class.

Class Implementation *Memory*

Instance Variables

Memory is implemented as a typedef Char *Memory*.

Memory

Superclasses

The *Memory* class is primitive and has no superclasses.

Messages and Responses

The *Memory* class is primitive and has no messages.

Class Variables

The *Memory* class has no class variables.

Function Listing In Alphabetic Order

Function Name	Description	Page	Scope	Macro
MemClear	Clears array elements	Mem - 2	Public	N
MemCopy	Copy array elements within the array	Mem - 3	Public	N
MemCutNSet	Delete elements, shift remaining	Mem - 5	Public	N
MemDestroy	Deallocate memory	Mem - 6	Public	N
MemDuplicate	Copy array into another	Mem - 7	Public	N
MemNew	Allocate memory	Mem - 8	Public	N
MemPasteNSet	Expand array, shift remaining	Mem - 9	Public	N
MemSetChr	Set elements to character	Mem - 10	Public	N

Function Listing By Scope

Public Functions

Function Name	Description	Page	Macro
MemClear	Clears array elements	Mem - 2	N
MemCopy	Copy array elements within the array	Mem - 3	N
MemCutNSet	Delete elements, shift remaining	Mem - 5	N
MemDestroy	Deallocate memory	Mem - 6	N
MemDuplicate	Copy array into another	Mem - 7	N
MemNew	Allocate memory	Mem - 8	N
MemPasteNSet	Expand array, shift remaining	Mem - 9	N
MemSetChr	Set elements to character	Mem - 10	N

Privat Functi ns

None			

Undocumented Functions

None			

Friend Functions

None			

Listing Of Functions by Category

Array Functions

Function Name	Description	Page	Scope	Macro
MemClear	Clears array elements	Mem - 2	Public	N
MemCopy	Copy array elements within the array	Mem - 3	Public	N
MemCutNSet	Delete elements, shift remaining	Mem - 5	Public	N
MemDuplicate	Copy array into another	Mem - 7	Public	N
MemPasteNSet	Expand array, shift remaining	Mem - 9	Public	N
MemSetChr	Set elements to character	Mem - 10	Public	N

Initialization Functions

Function Name	Description	Page	Scope	Macro
MemDestroy	Deallocate memory	Mem - 6	Public	N
MemNew	Allocate memory	Mem - 8	Public	N

Listing Of Functions With Macros Available

None			

This page intentionally left blank

Class Description for *MetaMessage*

Structure Name: MetaMessage

Abbreviation: Mms

Class Type: Primitive Class

Introduction

A *MetaMessage* is a definition of a message. MetaMessages allow a *MetaClass* to create an instance of *Message* which describes the final mapping of a message to a function.

Instances of *MetaMessage* are always declared and initialized as part of an array. The array is associated with an instance of *MetaClass*.

MetaClass, along with *Class*, *Object*, *Message*, *MetaSuperClass*, *MetaMessage*, and *Block* collectively define and implement the object-oriented properties of inheritance and dynamic binding (messaging) in C+O class libraries. Encapsulation, while supported by the above code, can be implemented by the programmer simply writing enough functions for a structure to obviate the need to access the structure members directly. These classes should be studied as a group to understand how C+O implements object-oriented C programs.

For further information, read the tutorial section and also study the class header files dllcls.h, lelcls.h, and trecls.h. These headers define instances of *MetaClass*, *MetaSuperClass* and *MetaMessage*.

MetaMessage

Description

An instance of *MetaMessage* describes the selector (name) for the message, the method to execute for this message, and the superclass (if any) this message was defined by.

The reader should famialiarize himself with the structure definition of *MetaMessage* since you will be initializing instances of *MetaMessage* statically.

The following rules apply when defining instances of *MetaMessage*:

1) If the method structure member is initialized to NULL and the superclass member is NULL, it means a subclass **must** implement the message.

2) If the method structure member is initialized to NULL and the superclass member is not NULL, then the method is inherited from the superclass named. In this case, the *MetaMessage* serves the purpose of propagating the message to a subclass which can override it if it so chooses.

3) If the method structure member is non-NULL and the superclass member is NULL it means the message originates from the class which owns this message. A subclass can override this message.

4) If the method structure member is non-NULL and the superclass member is non-NULL it means the superclass method is being overridden, not only for this class, but the superclass being overriden. This occurs for each level of superclass where the message was being overridden. A subclass can override this message.

5) If the method structure member is equal to the constant -1, it means we do not want to allow a subclass to override this message. Following this *MetaMessage* should be one which actually defines the message.

8) If the same message selector is defined by two or more superclasses, you should define a *MetaMessage* for each one. In effect, both are handled independently. This points out the advantage of an explicit inheritance mechanism where you must explicitly inherit messages from a superclass rather than have them "appear" automatically.

9) If a subclass overrides a message, any following messages with the same selector are also overriden IF they have the same implementation, i.e., if the method structure member is identical.

10) If there is more than one occurrence of a selector in the list of messages (because of identical selectors from different superclasses), the one nearest the top of the array will have priority over the others. This comes into play when looking up a message based on its selector string.

Glossary and Special Terms

No special glossary exists for *MetaMessage*. Instead, refer to the tutorial section on object-oriented techniques.

Class Implementation *MetaMessage*

Instance Variables

The following is the data structure definition for *MetaMessage*.

```
struct MetaMessage {    MediumInt    mmsCheck;
                        PSTR         superClsName;
                        PSTR         selector;
                        PMTH         method; }
```

mmsCheck	Uniquely defines *MetaMessage* structures.
superClsName	Name of the superclass which also has this message
selector	Name of the message.
method	The function which (optionally) overrides the superclass method.

Superclasses

MetaMessage is a primitive class and has no superclasses.

Messages and Responses

MetaMessage is a primitive class and therefore has no messages.

Class Variables

MetaMessage has no class variables.

Function Listing In Alphabetic Order

Function Name	Description	Page	Scope	Macro
MmsGetMethod	Get the method	Mms - 2	Public	N
MmsGetSelector	Get the name of selector	Mms - 6	Public	N
MmsGetSuper	Get name of superclass	Mms - 4	Public	N
MmsPrint	Display the instance	Mms - 7	Public	N
MmsValidate	Validate the instance	Mms - 8	Public	N

Function Listing By Scope

Public Functions

Function Name	Description	Page	Macro
MmsGetMethod	Get the method	Mms - 2	N
MmsGetSelector	Get the name of selector	Mms - 6	N
MmsGetSuper	Get name of superclass	Mms - 4	N
MmsPrint	Display the instance	Mms - 7	N
MmsValidate	Validate the instance	Mms - 8	N

Private Functions

None			

Undocumented Functions

None			

Friend Functions

None			

Listing Of Functions by Category

Query Functions

Function Name	Description	Page	Scope	Macro
MmsGetMethod	Get the method	Mms - 2	Public	N
MmsGetSelector	Get the name of selector	Mms - 6	Public	N
MmsGetSuper	Get name of superclass	Mms - 4	Public	N

Debug Functions

Function Name	Description	Page	Scope	Macro
MmsPrint	Display the instance	Mms - 7	Public	N
MmsValidate	Validate the instance	Mms - 8	Public	N

Listing Of Functions With Macros Available

None			

This page intentionally left blank

Class Description for *MetaSuperClass*

Structure Name: MetaSuperClass

Abbreviation: Msc

Class Type: Primitive Class

Introduction

The class *MetaSuperClass* is used to describe the superclasses from which a class inherits from. Instances of *MetaSuperClass* are always declared and initialized as part of an array.

Instances of *MetaSuperClass* are always initialized statically and is referenced by a single instance of *MetaClass*.

MetaClass, along with *Class*, *Object*, *Message*, *MetaSuperClass*, *MetaMessage*, and *Block* collectively define and implement the object-oriented properties of inheritance and dynamic binding (messaging) in C+O class libraries. Encapsulation, while supported by the above code, can be implemented by the programmer simply writing enough functions for a structure to obviate the need to access the structure members directly. These classes should be studied as a group to understand how C+O implements object-oriented C programs.

For further information, read the tutorial section and also study the class header files dllcls.h, lelcls.h, and trecls.h. These headers define instances of *MetaClass*, *MetaSuperClass* and *MetaMessage*.

MetaSuperClass

Description

An instance of *MetaSuperClass* describes the name of a superclass, the type of the superclass, and the offset of the superclass from its subclass.

When initializing an instance of *MetaSuperClass*, the name of each *MetaSuperClass* should be distinct from all other instances belonging to the *MetaClass*.

Glossary and Special Terms

There is no special glossary for *MetaSuperClass*. Instead, refer to the tutorial section on object-oriented techniques in C+O.

Class Implementation *MetaSuperClass*

Instance Variables

The following is the data structure definition for a *Class*.

```
struct MetaSuperClass {    MediumInt     mscCheck;
                           PSTR          name;
                           PMCL          mcl;
                           POBJ          subInstance;
                           POBJ          superInstance;
                           MediumInt     *offset; }
```

mscCheck	Uniquely identifies *MetaSuperClass* instances.
name	Name of superclass (overrides *MetaClass* name).
mcl	*MetaClass* which describes the class being inherited.
subInstance	An example instance of the superobject.
superInstance	An example instance of the object.
offset	A pointer to MediumInt where the superclass offset should be stored.

Superclasses

MetaSuperClass is a primitive class and has no superclasses

Messages and Responses

MetaSuperClass is a primitive class and has no messages

Class Variables

MetaSuperClass has no class variables.

Function Listing In Alphabetic Order

Function Name	Description	Page	Scope	Macro
MscGetMetaClass	Get the *MetaClass*	Msc - 3	Public	N
MscGetName	Get the name of superclass	Msc - 2	Public	N
MscGetOffset	Get offset of superclass	Msc - 4	Public	N
MscPrint	Display the contents of the *MetaSuperClass*	Msc - 5	Public	N
MscValidate	Validate that the instance is valid	Msc - 6	Public	N

Function Listing By Scope

Public Functions

Function Name	Description	Page	Macro
MscGetMetaClass	Get the *MetaClass*	Msc - 3	N
MscGetName	Get the name of superclass	Msc - 2	N
MscGetOffset	Get offset of superclass	Msc - 4	N
MscPrint	Display the contents of the *MetaSuperClass*	Msc - 5	N
MscValidate	Validate that the instance is valid	Msc - 6	N

Private Functions

None			

Undocumented Functions

None			

Friend Functions

None			

Listing Of Functions by Category

Query Functions

Function Name	Description	Page	Scope	Macro
MscGetMetaClass	Get the *MetaClass*	Msc - 3	Public	N
MscGetName	Get the name of superclass	Msc - 2	Public	N
MscGetOffset	Get offset of superclass	Msc - 4	Public	N

Debug Functions

Function Name	Description	Page	Scope	Macro
MscPrint	Display the contents of the *MetaSuperClass*	Msc - 5	Public	N
MscValidate	Validate that the instance is valid	Msc - 6	Public	N

Listing Of Functions With Macros Available

None			

Class Description for *Message*

Structure Name: Message

Abbreviation: Msg

Class Type: Primitive Class

Introduction

The *Message Class* is responsible for sending messages to an object and passing the object client to the function. This class is used only by the class *Class*. You will never need to use this class unless you are writing *Class* functions.

MetaClass, along with *Class*, *Object*, *Message*, *MetaSuperClass*, *MetaMessage*, and *Block* collectively define and implement the object-oriented properties of inheritance and dynamic binding (messaging) in C+O class libraries. Encapsulation, while supported by the above code, can be implemented by the programmer simply writing enough functions for a structure to obviate the need to access the structure members directly. These classes should be studied as a group to understand how C+O implements object-oriented C programs.

Description

When an object wishes to send a message, it identifies it by a message index or a message selector (name). The class then selects the appropriate message to route the message send to. Finally, an instance of *Message* invokes the proper method with a *Block* instance and passes an *Object* pointer.

Message

Glossary and Special Terms

There is no special glossary for *Message*. Instead, refer to the tutorial section on object-oriented techniques in C+O.

Class Implementation *Message*

Instance Variables

The following is the data structure definition for an *Object*

```
struct Message {        MediumInt   msgCheck;
                        PCLS        cls;
                        PMSG        superMsg;
                        PCLS        sub;
                        PSTR        selector;
                        MediumInt   offset;
                        PMTH        method; }
```

msgCheck	Uniquely identifies instances of *Message*
cls	*Class* which implements this message
superMsg	*Message* which is being inherited/overridden
sub	*Class* which implements this selector
selector	*String* which identifies the message
offset	*Object* offset which will yield an object of the right class for this message
method	Function which implements this message

Superclasses

The *Message* class is primitive and has no superclasses.

Messages and Responses

The *Message* class is primitive and has no messages.

Class Variables

The *Message* class has no class variables.

Function Listing In Alphabetic Order

Function Name	Description	Page	Scope	Macro
MsgDeInit	Deinit instance	Msg - 2	Public	N
MsgGetMethod	Return method	Msg - 4	Public	N
MsgGetOffset	Return the object offset	Msg - 3	Public	N
MsgGetSelector	Return selector	Msg - 5	Public	N
MsgInit	Initialize instanc	Msg - 6	Public	N
MsgPrint	Print the instance	Msg - 7	Public	N
MsgSend	Send message to object	Msg - 8	Public	N
MsgSendReturnInt	Send message to object, return int	Msg - 9	Public	N
MsgSendReturnPtr	Send message to object, return pointer	Msg - 10	Public	N
MsgSetSuperOffsetAndMethod	Override the method	N/A	Undoc	N

Function Listing By Scope

Public Functions

Function Name	Description	Page	Macro
MsgDeInit	Deinit instance	Msg - 2	N
MsgGetMethod	Return method	Msg - 4	N
MsgGetOffset	Return the object offset	Msg - 3	N
MsgGetSelector	Return selector	Msg - 5	N
MsgInit	Initialize instanc	Msg - 6	N
MsgPrint	Print the instance	Msg - 7	N
MsgSend	Send message to object	Msg - 8	N
MsgSendReturnInt	Send message to object, return int	Msg - 9	N
MsgSendReturnPtr	Send message to object, return pointer	Msg - 10	N

Private Functions

None			

Undocumented Functions

Function Name	Description	Page	Macro
MsgSetSuperOffsetAndMethod	Override the method	N/A	N

Friend Functions

None			

Listing Of Functions by Category

Initialization Functions

Function Name	Description	Page	Scope	Macro
MsgDeInit	Deinit instance	Msg - 2	Public	N
MsgInit	Initialize instanc	Msg - 6	Public	N

Query Functions

Function Name	Description	Page	Scope	Macro
MsgGetMethod	Return method	Msg - 4	Public	N
MsgGetOffset	Return the object offset	Msg - 3	Public	N
MsgGetSelector	Return selector	Msg - 5	Public	N

Debug Functions

Function Name	Description	Page	Scope	Macro
MsgPrint	Print the instance	Msg - 7	Public	N

Object Functions

Function Name	Description	Page	Scope	Macro
MsgSend	Send message to object	Msg - 8	Public	N
MsgSendReturnInt	Send message to object, return int	Msg - 9	Public	N
MsgSendReturnPtr	Send message to object, return pointer	Msg - 10	Public	N

Undocumented Functi ns

Function Name	Description	Page	Scope	Macro
MsgSetSuperOffsetAndMethod	Override the method	N/A	Undoc	N

Listing Of Functions With Macros Available

None			

This page intentionally left blank

Class Description for *Object*

Structure Name: Object

Abbreviation: Obj

Class Type: Primitive Class.

Introduction

The *Object* class is the conduit by which a structure gains the object-oriented properties of inheritance and dynamic binding. Any class which desires these properties must include an instance of *Object* as the first structure member.

A structure which includes *Object* can inherit properties and messages from other object-oriented classes. It can be inherited by other other object-oriented classes. Finally, it can send messages which can be received and implemented by subclasses (clients) or superclasses.

A class uses *Object* to retrieve client pointers, send messages, and ask questions about the type of the object.

Instances of object are never created directly. Classes which include *Object* in their definition should always create instances via ClsCreateObject and destroy instances through ObjDestroy. A class need not initialize an *Object* (ObjInit) as this will occur automatically via ClsCreateObject.

MetaClass, along with *Class*, *Object*, *Message*, *MetaSuperClass*, *MetaMessage*, and *Block* collectively define and implement the object-oriented properties of inheritance and dynamic binding (messaging) in C+O class libraries. Encapsulation, while supported by the above code, can be implemented by the programmer simply writing enough functions for a structure to obviate the need to access the structure members directly. These classes should be studied as a group to understand how C+O implements object-oriented C programs.

Object

Description

The function of the class *Object* is mainly to keep track of the instance of *Class* that is associated with this structure. The *Class* instance enables the *Object* instance to determine whether or not its part of a larger object and what messages it can respond to (and what method to call when a message is invoked).

Glossary and Special Terms

A **subobject** is what a subclass is to a class. It is the object instance which incorporates the object.

A **superobject** is what a superclass is to a class. It is an object instance contained by an object.

The **root object** is an object which has no subobject. Root objects are typically created by ClsCreateObject and destroyed by ObjDestroy.

For other definitions, see the tutorial section.

Class Implementation *Object*

Superclasses

The *Object* class is primitive and has no superclasses.

Messages and Responses

The *Object* class is primitive and has no messages.

Class Variables

The *Object* class has no class variables.

Function Listing In Alphabetic Order

Function Name	Description	Page	Scope	Macro
ObjDeInit	Deinitialize instance	Obj - 2	Public	N
ObjDestroy	Deinitialize and deallocate object	Obj - 3	Public	N
ObjGetClient	Return a client	Obj - 4	Public	N
ObjGetClientOrNull	Return client if non-NULL	Obj - 6	Public	N
ObjGetCls	Return class	Obj - 8	Public	N
ObjGetClsName	Return class name	Obj - 9	Public	N
ObjGetImmediateClient	Return immediate client	Obj - 10	Public	N
ObjGetMethodAndOffset	Return method and client offset	Obj - 11	Public	N
ObjGetNthSuperObject	Return nth superobject	Obj - 13	Public	N
ObjGetRootClient	Get root client	Obj - 14	Public	N
ObjGetRootClientSize	Get root object size	Obj - 15	Public	N
ObjGetSize	Get object size	Obj - 16	Public	N
ObjGetSubObjectOffset	Return offset to immediate subobject	Obj - 17	Public	N
ObjInit	Initialize object	Obj - 18	Public	N
ObjIsRoot	Return Treu if is root subobject	Obj - 19	Public	N
ObjRespondsToSelector	Does object understand message?	Obj - 20	Public	N
ObjSendMessage	Send message to object	Obj - 21	Public	N
ObjSendMessageReturnInt	Send message to object, return Int	Obj - 22	Public	N
ObjSendMessageReturnPtr	Send message to object, return pointer	Obj - 23	Public	N

Function Listing By Scope

Public Functions

Function Name	Description	Page	Macro
ObjDeInit	Deinitialize instance	Obj - 2	N
ObjDestroy	Deinitialize and deallocate object	Obj - 3	N
ObjGetClient	Return a client	Obj - 4	N
ObjGetClientOrNull	Return client if non-NULL	Obj - 6	N
ObjGetCls	Return class	Obj - 8	N
ObjGetClsName	Return class name	Obj - 9	N
ObjGetImmediateClient	Return immediate client	Obj - 10	N
ObjGetMethodAndOffset	Return method and client offset	Obj - 11	N
ObjGetNthSuperObject	Return nth superobject	Obj - 13	N
ObjGetRootClient	Get root client	Obj - 14	N
ObjGetRootClientSize	Get root object size	Obj - 15	N
ObjGetSize	Get object size	Obj - 16	N
ObjGetSubObjectOffset	Return offset to immediate subobject	Obj - 17	N
ObjInit	Initialize object	Obj - 18	N
ObjIsRoot	Return Treu if is root subobject	Obj - 19	N

Public Functions (cont)

Function Name	Description	Page	Macro
ObjRespondsToSelector	Does object understand message?	Obj - 20	N
ObjSendMessage	Send message to object	Obj - 21	N
ObjSendMessageReturnInt	Send message to object, return Int	Obj - 22	N
ObjSendMessageReturnPtr	Send message to object, return pointer	Obj - 23	N

Private Functions

None			

Undocumented Functions

None			

Friend Functions

None			

Listing Of Functions by Category

Initialization Functions

Function Name	Description	Page	Scope	Macro
ObjDeInit	Deinitialize instance	Obj - 2	Public	N
ObjDestroy	Deinitialize and deallocate object	Obj - 3	Public	N
ObjInit	Initialize object	Obj - 18	Public	N

Client Functions

Function Name	Description	Page	Scope	Macro
ObjGetClient	Return a client	Obj - 4	Public	N
ObjGetClientOrNull	Return client if non-NULL	Obj - 6	Public	N
ObjGetImmediateClient	Return immediate client	Obj - 10	Public	N
ObjGetNthSuperObject	Return nth superobject	Obj - 13	Public	N
ObjGetRootClient	Get root client	Obj - 14	Public	N
ObjGetSubObjectOffset	Return offset to immediate subobject	Obj - 17	Public	N

Query Functions

Function Name	Description	Page	Scope	Macro
ObjGetCls	Return class	Obj - 8	Public	N
ObjGetClsName	Return class name	Obj - 9	Public	N
ObjGetRootClientSize	Get root object size	Obj - 15	Public	N
ObjGetSize	Get object size	Obj - 16	Public	N
ObjIsRoot	Return Treu if is root subobject	Obj - 19	Public	N

Query Functions

Function Name	Description	Page	Scope	Macro
ObjGetMethodAndOffset	Return method and client offset	Obj - 11	Public	N

Messaging Functions

Function Name	Description	Page	Scope	Macro
ObjRespondsToSelector	Does object understand message?	Obj - 20	Public	N
ObjSendMessage	Send message to object	Obj - 21	Public	N
ObjSendMessageReturnInt	Send message to object, return Int	Obj - 22	Public	N
ObjSendMessageReturnPtr	Send message to object, return pointer	Obj - 23	Public	N

Listing Of Functions With Macros Available

None			

This page intentionally left blank

Class Description for *String*

Structure Name: String

Abbreviation: Str

Class Type: Primitive.

Description

The *String* class is used to represent and manipulate Null terminated character arrays and are identical to "normal" C strings.

Some functions have been provided in this class which are available as functions already from the C compiler manufacturer. We have provided these not to insult anyone but to provide a function which maps to our naming convention and is perhaps more descriptive. We recognize this class is somewhat thin and future updates will expand the functions in this class.

Glossary and Special Terms

None

Class Implementation *String*

Instance Variables

The *String* class is implemented as a typedef Char *String*.

String

Superclasses

The *String* class does not inherit from any other classes.

Class Variables

The *String* class has no class instance variables.

Function Listing In Alphabetic Order

Function Name	Description	Page	Scope	Macro
StrBasicExtract	Extract a string	N/A	Undoc	N
StrExtract	Extract string as specified	Str - 2	Public	N
StrFromDate	Fill string with a date	Str - 4	Public	N
StrFromMediumInt	Integer to string	Str - 6	Public	N
StrInit	Init string to zero	Str - 8	Public	Y
StrReplaceSubStr	Replace sub-string in string	Str - 9	Public	N
StrSet	Copy string to another	Str - 11	Public	Y
StrSqueeze	Removes any character from string	Str - 13	Public	N
StrToDate	Parse a date string for year, month, day	Str - 15	Public	N
StrToLower	Change case of string to lower	Str - 17	Public	Y
StrToMediumInt	*String* to integer	Str - 19	Public	N
StrToUpper	Change case of string to upper	Str - 21	Public	Y

Function Listing By Scope

Public Functions

Function Name	Description	Page	Macro
StrExtract	Extract string as specified	Str - 2	Y
StrFromDate	Fill string with a date	Str - 4	Y
StrFromMediumInt	Integer to string	Str - 6	Y
StrInit	Init string to zero	Str - 8	Y
StrReplaceSubStr	Replace sub-string in string	Str - 9	Y
StrSet	Copy string to another	Str - 11	Y
StrSqueeze	Removes any character from string	Str - 13	Y
StrToDate	Parse a date string for year, month, day	Str - 15	Y
StrToLower	Change case of string to lower	Str - 17	Y
StrToMediumInt	*String* to integer	Str - 19	Y
StrToUpper	Change case of string to upper	Str - 21	Y

Private Functi ns

None			

Undocumented Functions

Function Name	Description	Page	Macro
StrBasicExtract	Extract a string	N/A	Y

Friend Functions

None			

Listing Of Functions by Category

Manipulation Functions

Function Name	Description	Page	Scope	Macro
StrBasicExtract	Extract a string	N/A	Undoc	N
StrExtract	Extract string as specified	Str - 2	Public	N
StrReplaceSubStr	Replace sub-string in string	Str - 9	Public	N
StrSet	Copy string to another	Str - 11	Public	Y
StrSqueeze	Removes any character from string	Str - 13	Public	N
StrToLower	Change case of string to lower	Str - 17	Public	Y

Conversion Functions

Function Name	Description	Page	Scope	Macro
StrFromDate	Fill string with a date	Str - 4	Public	N
StrFromMediumInt	Integer to string	Str - 6	Public	N
StrToDate	Parse a date string for year, month, day	Str - 15	Public	N
StrToMediumInt	*String* to integer	Str - 19	Public	N
StrToUpper	Change case of string to upper	Str - 21	Public	Y

Initialization Functions

Function Name	Description	Page	Scope	Macro
StrInit	Init string to zero	Str - 8	Public	Y

Listing Of Functions With Macros Available

Function Name	Description	Page	Scope
StrInit	Init string to zero	Str - 8	Public
StrSet	Copy string to another	Str - 11	Public
StrToLower	Change case of string to lower	Str - 17	Public
StrToUpper	Change case of string to upper	Str - 21	Public

Class Description for *Tree*

Structure Name: Tree

Abbreviation: Tre

Class Type: Inheritable class

Introduction

The *Tree* structure is organized so that items of information are related hierarchically by branches. This data structure is important for many applications such as outlining, bill of materials, parsing, and genetics.

Description

A tree is a finite set of one or more nodes such that: (i) there is a specially designated node called the root; (ii) the remaining nodes are partitioned into disjoint sets where each of these sets is a tree. This is a recursive definition and will be used in describing many of the functions throughout the reference section for the *Tree* class.

Figure 1 shows a tree with 13 nodes. The root of the tree is node A.

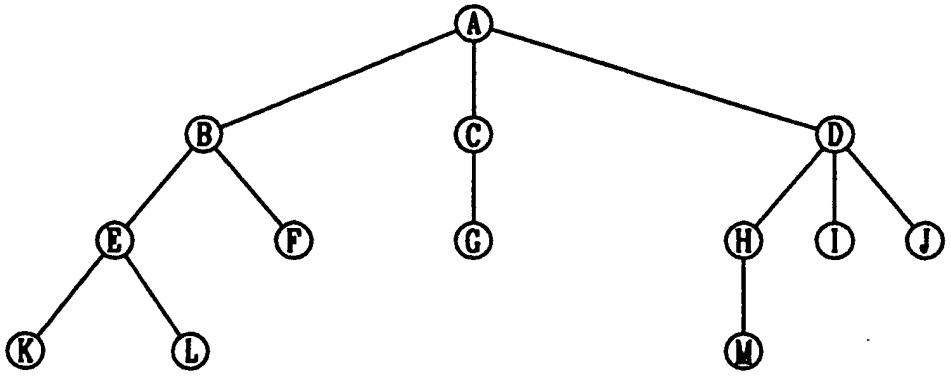


Figure 1

Glossary and Special Terms

There are many terms which are used when referring to trees. The following are the definitions we ascribe to them.

An instance of a *Tree* (or its subclass) is often called a **node**. It contains the branches to other trees.

The number of **subtrees** of a tree is called its **degree**. In Figure 2 the degree of A is 3, of C is 1 and of F is zero.

Trees of degree zero are called **leaves**. In Figure 2 K, L and M are some of the leaf nodes.

Trees with degree greater than zero are called **branches**. In Figure 2 A, B, E, C, D, and H are some of the branches.

The subtrees of a *Tree* X are referred to as the **children** of X. X is referred to as the **parent** *Tree*. In Figure 2 the children of D are H, I, J; the parent of D is A.

A tree with no parent is called a **root**. A is the root in Figure 2.

Children of the same parent are said to be **siblings**. In Figure 2 H, I and J are siblings.

For a tree with children, there is a designated **first** child and a **last** child. A tree with one child has the same tree for the first child as its last child.

A **successor** or **next** sibling of a tree is the immediate sibling which leads toward the last child of their common parent. A **predecessor** or **previous** sibling of a tree is the immediate sibling leading toward the first child of their common parent.

The **ancestors** of a tree are all the trees along the path from the root to that tree. In Figure 2 the ancestors of M are A, D and H.

The **descendants** of a tree are its children and their children and so on. In Figure 2 the descendants of B are E, K, L, and F.

The siblings of a subtree tree X, are the **uncles** of the children of X. In Figure 2 the trees B and C are uncles of H, I and J.

The **sequential predecessor/ successor** of a tree X is found by taking (i) the first child if any; (ii) or the adjacent predecessor/successor sibling if any; (iii) or the adjacent uncle. In this way, the whole tree is traversed once (in pre-order).

A **range** of trees is defined by a **beginning** tree and an **ending** tree. Both trees must be siblings. The beginning tree must precede the ending tree. The range includes any and all elements between the beginning and ending tree.

Examples of valid ranges in Figure 2:

1) B through D is the subtrees B, C, and D. 2) H through H is the subtree H

Examples of invalid ranges in in Figure 2:

1) E through B. E and B are not siblings. 2) F through G. F and G are not siblings. 3) J through H. J does not precede H.

By **walking** a tree, we mean we **visit** a subset of the nodes of the tree in a particular order. Visiting a node means to call a function, passing that node a pointer to the tree or one of its clients.

There are many ways to walk a tree. An **in-order walk** of X is defined as performing an in-order walk of the children of X then visiting X. In Figure 2 an in-order walk of B would visit the nodes K, L, E, F and B in that order.

A **pre-order walk** of X is defined as visiting X then performing a pre-order walk of the children of X. In Figure 2 a pre-order walk of B would visit the nodes E, F, K, and L in that order.

The **level** of a tree is defined by initially letting the root be at level L, then its children are at level L+1 and so on. Figure 2 shows the levels of the trees.

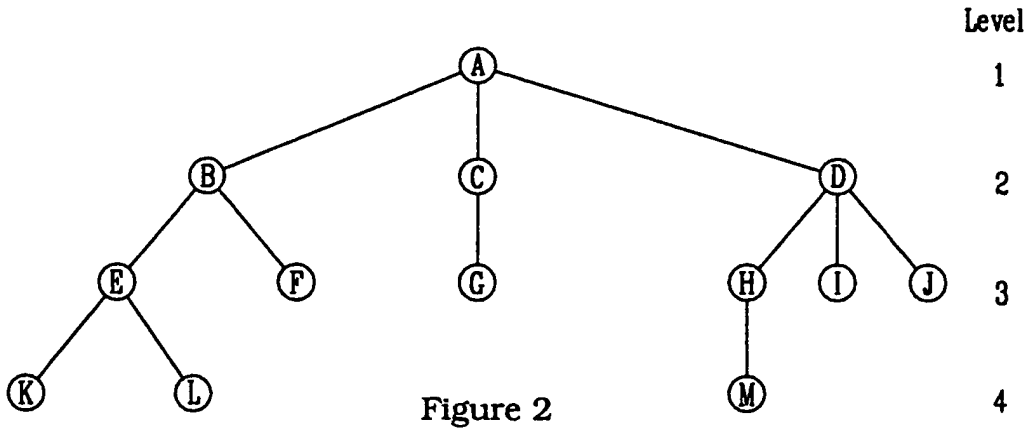


Figure 2

Class Implementation *Tree*

Instance Variables

The following is the data structure definition for a tree.

```
struct Tree {        Obj          obj;
                     MediumInt    treCheck;
                     Dll          children;
                     Lel          siblings;  }
```

obj	Gives *Tree* class object-oriented properties of inheritance and dynamic binding.
treCheck	Uniquely identifies *Tree* instances.
children	*List* of children.
siblings	*ListElement* in a list of siblings.

Superclasses

Type	Name	Description
List	Dll	Children of tree
ListElement	Lel	Siblings

Messages and Responses

Tree overrides the destroy message from its *List* superclass and its *ListElement* superclass. The implementation of the destroy message is handled by TreDestroy. Subclasses of *Tree* should override this message.

Selector	Method	Description
destroy	TreDestroy	Destroy the object

Class Variables

Type	Name	Description
Mcl	TreMcl	Metaclass description for *Tree*.
MediumInt	TreDllOffset	Client offset of *Tree* from superclass Dll
MediumInt	TreLelOffset	Client offset of *Tree* from superclass Lel

Function Listing In Alphabetic Order

Function Name	Description	Page	Scope	Macro
TreAsDll	Return node as list	Tre - 2	Private	Y
TreAsLel	Return node as list element	Tre - 3	Private	Y
TreAsObj	Return node as object	Tre - 4	Private	Y
TreClear	Clear the tree	Tre - 5	Public	N
TreClient	Return client of node	Tre - 6	Public	Y
TreClientFindChild	Visit search function: children	Tre - 7	Public	N
TreClientFirstChild	Return first child node as client	Tre - 9	Public	Y
TreClientLastChild	Return last child node as client	Tre - 11	Public	Y
TreClientLastLeaf	Return last leaf node as client	Tre - 13	Public	Y
TreClientNext	Return next node as client	Tre - 15	Public	Y
TreClientNextPreOrder	Return next PreOrder client node	Tre - 17	Public	Y
TreClientNextUncle	Return next uncle as client	Tre - 19	Public	Y
TreClientParent	Return parent node	Tre - 21	Public	Y
TreClientPrev	Return prev client	Tre - 23	Public	Y
TreClientPrevPreOrder	Return previous client PreOrderly	Tre - 25	Public	Y
TreClientVisitBranchInOrder	Visit function: branch in-order	Tre - 27	Public	N
TreClientVisitChildren	Visit function: all children	Tre - 29	Public	N
TreClientVisitChildrenBwd	Visit function: all children	Tre - 31	Public	N
TreClientVisitDescBranchInOrder	Visit function: descendents	Tre - 33	Public	N
TreClientVisitDescInOrder	Visit function: descendents	Tre - 35	Public	N
TreClientVisitDescInOrderBwd	Visit function: descendents	Tre - 37	Public	N
TreClientVisitDescLeaves	Visit function: descendents	Tre - 39	Private	N
TreClientVisitDescPreOrder	Visit function: descendents	Tre - 41	Public	N
TreClientVisitInOrder	Visit function: in-order	Tre - 43	Public	N
TreClientVisitInOrderBwd	Visit function: in-order	Tre - 45	Public	N
TreClientVisitLeaves	Visit function: leaves	Tre - 47	Public	N
TreClientVisitParents	Visit function: nearest parents first	Tre - 49	Public	N
TreClientVisitPreOrder	Visit function: pre-order	Tre - 51	Public	N
TreClientVisitRange	Visit function: range	Tre - 53	Public	N
TreClientVisitSuccPreOrder	Visit function: all successors	Tre - 55	Public	N
TreClientVisitSuccessors	Visit function: successors	Tre - 57	Public	N
TreCutChildren	Cut children from tree	Tre - 59	Public	Y
TreCutRange	Cut node(s) from tree	Tre - 61	Public	Y
TreDeInit	Deinitialize *Tree* object	Tre - 63	Public	N
TreDestroy	Deinitialize *Tree* object and free space	Tre - 64	Public	N
TreDestroyChildren	Destroy any children of a tre	Tre - 65	Public	N
TreFirstChild	Return first child	Tre - 66	Private	Y
TreHasChildren	Does node have any children	Tre - 68	Public	Y
TreHasSiblings	Does node have any siblings	Tre - 69	Public	Y
TreInit	Initialize tree object	Tre - 70	Public	N
TreIsChild	Does the node have a parent	Tre - 71	Public	Y
TreIsDirectAncestor	Is node a direct ancester	Tre - 72	Public	N

Functi n Listing In Alphabetic Order (cont)

Function Name	Description	Page	Scope	Macro
TreIsRoot	Does the node have no parent	Tre - 73	Public	Y
TreLastChild	Return last child	Tre - 74	Private	Y
TreLastLeaf	Return last leaf	Tre - 76	Private	N
TreNext	Return next node	Tre - 78	Private	Y
TreNextPreOrder	Return next node PreOrderly	Tre - 80	Private	N
TreNextUncle	Return next uncle	Tre - 82	Private	N
TreParent	Return parent node	Tre - 84	Private	Y
TrePasteRangeAfterSibling	Paste range of siblings	Tre - 86	Public	Y
TrePasteRangeBeforeSibling	Paste range of siblings	Tre - 88	Public	Y
TrePasteRangeFirstChild	Paste children	Tre - 90	Public	Y
TrePasteRangeLastChild	Paste children	Tre - 92	Public	Y
TrePrev	Return previous node	Tre - 94	Private	Y
TrePrevPreOrder	Return previous node PreOrderly	Tre - 96	Private	N
TreSendDestroy	Send message for tree destruction	Tre - 98	Public	N
TreVisitBranchInOrder	Visit function: branch in-order	Tre - 99	Private	N
TreVisitChildren	Visit function: children	Tre - 101	Private	N
TreVisitChildrenBwd	Visit function: children	Tre - 103	Private	N
TreVisitDescBranchInOrder	Visit function: descendents	Tre - 105	Private	N
TreVisitDescInOrder	Visit function: descendents	Tre - 107	Private	N
TreVisitDescInOrderBwd	Visit function: descendents	Tre - 109	Private	N
TreVisitDescPreOrder	Visit function: descendents	Tre - 111	Private	N
TreVisitInOrder	Visit function: in-order	Tre - 113	Private	N
TreVisitInOrderBwd	Visit function: in-order	Tre - 115	Private	N
TreVisitLeaves	Visit function: leaves	Tre - 117	Private	N
TreVisitParents	Visit function: nearest parents first	Tre - 119	Private	N
TreVisitPreOrder	Visit function: pre-order	Tre - 121	Private	N
TreVisitRange	Visit function: range	Tre - 123	Private	N
TreVisitSuccPreOrder	Visit function: all successors	Tre - 125	Private	N
TreVisitSuccessors	Visit function: successors	Tre - 127	Private	N

Function Listing By Scope

Public Functions

Function Name	Description	Page	Macro
TreClear	Clear the tree	Tre - 5	N
TreClient	Return client of node	Tre - 6	N
TreClientFindChild	Visit search function: children	Tre - 7	N
TreClientFirstChild	Return first child node as client	Tre - 9	N
TreClientLastChild	Return last child node as client	Tre - 11	N
TreClientLastLeaf	Return last leaf node as client	Tre - 13	N
TreClientNext	Return next node as client	Tre - 15	N
TreClientNextPreOrder	Return next PreOrder client node	Tre - 17	N
TreClientNextUncle	Return next uncle as client	Tre - 19	N
TreClientParent	Return parent node	Tre - 21	N
TreClientPrev	Return prev client	Tre - 23	N
TreClientPrevPreOrder	Return previous client PreOrderly	Tre - 25	N
TreClientVisitBranchInOrder	Visit function: branch in-order	Tre - 27	N
TreClientVisitChildren	Visit function: all children	Tre - 29	N
TreClientVisitChildrenBwd	Visit function: all children	Tre - 31	N
TreClientVisitDescBranchInOrder	Visit function: descendents	Tre - 33	N
TreClientVisitDescInOrder	Visit function: descendents	Tre - 35	N
TreClientVisitDescInOrderBwd	Visit function: descendents	Tre - 37	N
TreClientVisitDescPreOrder	Visit function: descendents	Tre - 41	N
TreClientVisitInOrder	Visit function: in-order	Tre - 43	N
TreClientVisitInOrderBwd	Visit function: in-order	Tre - 45	N
TreClientVisitLeaves	Visit function: leaves	Tre - 47	N
TreClientVisitParents	Visit function: nearest parents first	Tre - 49	N
TreClientVisitPreOrder	Visit function: pre-order	Tre - 51	N
TreClientVisitRange	Visit function: range	Tre - 53	N
TreClientVisitSuccPreOrder	Visit function: all successors	Tre - 55	N
TreClientVisitSuccessors	Visit function: successors	Tre - 57	N
TreCutChildren	Cut children from tree	Tre - 59	N
TreCutRange	Cut node(s) from tree	Tre - 61	N
TreDeInit	Deinitialize *Tree* object	Tre - 63	N
TreDestroy	Deinitialize *Tree* object and free space	Tre - 64	N
TreDestroyChildren	Destroy any children of a tre	Tre - 65	N
TreHasChildren	Does node have any children	Tre - 68	N
TreHasSiblings	Does node have any siblings	Tre - 69	N
TreInit	Initialize tree object	Tre - 70	N
TreIsChild	Does the node have a parent	Tre - 71	N
TreIsDirectAncestor	Is node a direct ancester	Tre - 72	N
TreIsRoot	Does the node have no parent	Tre - 73	N
TrePasteRangeAfterSibling	Paste range of siblings	Tre - 86	N

Public Functions (cont)

Function Name	Description	Page	Macro
TrePasteRangeBeforeSibling	Paste range of siblings	Tre - 88	N
TrePasteRangeFirstChild	Paste children	Tre - 90	N
TrePasteRangeLastChild	Paste children	Tre - 92	N
TreSendDestroy	Send message for tree destruction	Tre - 98	N

Private Functions

Function Name	Description	Page	Macro
TreAsDll	Return node as list	Tre - 2	N
TreAsLel	Return node as list element	Tre - 3	N
TreAsObj	Return node as object	Tre - 4	N
TreClientVisitDescLeaves	Visit function: descendents	Tre - 39	N
TreFirstChild	Return first child	Tre - 66	N
TreLastChild	Return last child	Tre - 74	N
TreLastLeaf	Return last leaf	Tre - 76	N
TreNext	Return next node	Tre - 78	N
TreNextPreOrder	Return next node PreOrderly	Tre - 80	N
TreNextUncle	Return next uncle	Tre - 82	N
TreParent	Return parent node	Tre - 84	N
TrePrev	Return previous node	Tre - 94	N
TrePrevPreOrder	Return previous node PreOrderly	Tre - 96	N
TreVisitBranchInOrder	Visit function: branch in-order	Tre - 99	N
TreVisitChildren	Visit function: children	Tre - 101	N
TreVisitChildrenBwd	Visit function: children	Tre - 103	N
TreVisitDescBranchInOrder	Visit function: descendents	Tre - 105	N
TreVisitDescInOrder	Visit function: descendents	Tre - 107	N
TreVisitDescInOrderBwd	Visit function: descendents	Tre - 109	N
TreVisitDescPreOrder	Visit function: descendents	Tre - 111	N
TreVisitInOrder	Visit function: in-order	Tre - 113	N
TreVisitInOrderBwd	Visit function: in-order	Tre - 115	N
TreVisitLeaves	Visit function: leaves	Tre - 117	N
TreVisitParents	Visit function: nearest parents first	Tre - 119	N
TreVisitPreOrder	Visit function: pre-order	Tre - 121	N
TreVisitRange	Visit function: range	Tre - 123	N
TreVisitSuccPreOrder	Visit function: all successors	Tre - 125	N
TreVisitSuccessors	Visit function: successors	Tre - 127	N

Undocum nted Functi ns

None			

Friend Functions

None			

Listing Of Functions by Category

Superclass Functions

Function Name	Description	Page	Scope	Macro
TreAsDll	Return node as list	Tre - 2	Private	Y
TreAsLel	Return node as list element	Tre - 3	Private	Y
TreAsObj	Return node as object	Tre - 4	Private	Y

Edit Functions

Function Name	Description	Page	Scope	Macro
TreClear	Clear the tree	Tre - 5	Public	N
TreCutChildren	Cut children from tree	Tre - 59	Public	Y
TreCutRange	Cut node(s) from tree	Tre - 61	Public	Y
TrePasteRangeAfterSibling	Paste range of siblings	Tre - 86	Public	Y
TrePasteRangeBeforeSibling	Paste range of siblings	Tre - 88	Public	Y
TrePasteRangeFirstChild	Paste children	Tre - 90	Public	Y
TrePasteRangeLastChild	Paste children	Tre - 92	Public	Y

Query Functions

Function Name	Description	Page	Scope	Macro
TreClient	Return client of node	Tre - 6	Public	Y
TreClientFindChild	Visit search function: children	Tre - 7	Public	N
TreClientFirstChild	Return first child node as client	Tre - 9	Public	Y
TreClientLastChild	Return last child node as client	Tre - 11	Public	Y
TreClientLastLeaf	Return last leaf node as client	Tre - 13	Public	Y
TreClientNext	Return next node as client	Tre - 15	Public	Y
TreClientNextPreOrder	Return next PreOrder client node	Tre - 17	Public	Y
TreClientNextUncle	Return next uncle as client	Tre - 19	Public	Y
TreClientParent	Return parent node	Tre - 21	Public	Y
TreClientPrev	Return prev client	Tre - 23	Public	Y
TreClientPrevPreOrder	Return previous client PreOrderly	Tre - 25	Public	Y
TreFirstChild	Return first child	Tre - 66	Private	Y
TreHasChildren	Does node have any children	Tre - 68	Public	Y
TreHasSiblings	Does node have any siblings	Tre - 69	Public	Y
TreIsChild	Does the node have a parent	Tre - 71	Public	Y
TreIsDirectAncestor	Is node a direct ancester	Tre - 72	Public	N
TreIsRoot	Does the node have no parent	Tre - 73	Public	Y
TreLastChild	Return last child	Tre - 74	Private	Y
TreLastLeaf	Return last leaf	Tre - 76	Private	N
TreNext	Return next node	Tre - 78	Private	Y
TreNextPreOrder	Return next node PreOrderly	Tre - 80	Private	N
TreNextUncle	Return next uncle	Tre - 82	Private	N
TreParent	Return parent node	Tre - 84	Private	Y
TrePrev	Return previous node	Tre - 94	Private	Y
TrePrevPreOrder	Return previous node PreOrderly	Tre - 96	Private	N

Visit Functi ns

Function Name	Description	Page	Scope	Macro
TreClientVisitBranchInOrder	Visit function: branch in-order	Tre - 27	Public	N
TreClientVisitChildren	Visit function: all children	Tre - 29	Public	N
TreClientVisitChildrenBwd	Visit function: all children	Tre - 31	Public	N
TreClientVisitDescBranchInOrder	Visit function: descendents	Tre - 33	Public	N
TreClientVisitDescInOrder	Visit function: descendents	Tre - 35	Public	N
TreClientVisitDescInOrderBwd	Visit function: descendents	Tre - 37	Public	N
TreClientVisitDescLeaves	Visit function: descendents	Tre - 39	Private	N
TreClientVisitDescPreOrder	Visit function: descendents	Tre - 41	Public	N
TreClientVisitInOrder	Visit function: in-order	Tre - 43	Public	N
TreClientVisitInOrderBwd	Visit function: in-order	Tre - 45	Public	N
TreClientVisitLeaves	Visit function: leaves	Tre - 47	Public	N
TreClientVisitParents	Visit function: nearest parents first	Tre - 49	Public	N
TreClientVisitPreOrder	Visit function: pre-order	Tre - 51	Public	N
TreClientVisitRange	Visit function: range	Tre - 53	Public	N
TreClientVisitSuccPreOrder	Visit function: all successors	Tre - 55	Public	N
TreClientVisitSuccessors	Visit function: successors	Tre - 57	Public	N
TreVisitBranchInOrder	Visit function: branch in-order	Tre - 99	Private	N
TreVisitChildren	Visit function: children	Tre - 101	Private	N
TreVisitChildrenBwd	Visit function: children	Tre - 103	Private	N
TreVisitDescBranchInOrder	Visit function: descendents	Tre - 105	Private	N
TreVisitDescInOrder	Visit function: descendents	Tre - 107	Private	N
TreVisitDescInOrderBwd	Visit function: descendents	Tre - 109	Private	N
TreVisitDescPreOrder	Visit function: descendents	Tre - 111	Private	N
TreVisitInOrder	Visit function: in-order	Tre - 113	Private	N
TreVisitInOrderBwd	Visit function: in-order	Tre - 115	Private	N
TreVisitLeaves	Visit function: leaves	Tre - 117	Private	N
TreVisitParents	Visit function: nearest parents first	Tre - 119	Private	N
TreVisitPreOrder	Visit function: pre-order	Tre - 121	Private	N
TreVisitRange	Visit function: range	Tre - 123	Private	N
TreVisitSuccPreOrder	Visit function: all successors	Tre - 125	Private	N
TreVisitSuccessors	Visit function: successors	Tre - 127	Private	N

Initialization Functions

Function Name	Description	Page	Scope	Macro
TreDeInit	Deinitialize *Tree* object	Tre - 63	Public	N
TreDestroy	Deinitialize *Tree* object and free space	Tre - 64	Public	N
TreDestroyChildren	Destroy any children of a tre	Tre - 65	Public	N
TreInit	Initialize tree object	Tre - 70	Public	N
TreSendDestroy	Send message for tree destruction	Tre - 98	Public	N

Listing Of Functions With Macros Available

Function Name	Description	Page	Scope
TreAsDll	Return node as list	Tre - 2	Private
TreAsLel	Return node as list element	Tre - 3	Private
TreAsObj	Return node as object	Tre - 4	Private
TreClient	Return client of node	Tre - 6	Public
TreClientFirstChild	Return first child node as client	Tre - 9	Public
TreClientLastChild	Return last child node as client	Tre - 11	Public
TreClientLastLeaf	Return last leaf node as client	Tre - 13	Public
TreClientNext	Return next node as client	Tre - 15	Public
TreClientNextPreOrder	Return next PreOrder client node	Tre - 17	Public
TreClientNextUncle	Return next uncle as client	Tre - 19	Public
TreClientParent	Return parent node	Tre - 21	Public
TreClientPrev	Return prev client	Tre - 23	Public
TreClientPrevPreOrder	Return previous client PreOrderly	Tre - 25	Public
TreCutChildren	Cut children from tree	Tre - 59	Public
TreCutRange	Cut node(s) from tree	Tre - 61	Public
TreFirstChild	Return first child	Tre - 66	Private
TreHasChildren	Does node have any children	Tre - 68	Public
TreHasSiblings	Does node have any siblings	Tre - 69	Public
TreIsChild	Does the node have a parent	Tre - 71	Public
TreIsRoot	Does the node have no parent	Tre - 73	Public
TreLastChild	Return last child	Tre - 74	Private
TreNext	Return next node	Tre - 78	Private
TreParent	Return parent node	Tre - 84	Private
TrePasteRangeAfterSibling	Paste range of siblings	Tre - 86	Public
TrePasteRangeBeforeSibling	Paste range of siblings	Tre - 88	Public
TrePasteRangeFirstChild	Paste children	Tre - 90	Public
TrePasteRangeLastChild	Paste children	Tre - 92	Public
TrePrev	Return previous node	Tre - 94	Private

Class Description for *Task*

Structure Name: Task

Abbreviation: Tsk

Class Type: Primitive Class

Introduction

The *Task* class models the program being executed. Currently its major usage is to handle exceptions that may arise during program execution. A task may set up any number of exception handlers for intercepting exceptions that are raised by a program. C+O class libraries include a debug library which checks parameter input to functions scrupulously, generating exceptions if the parameter is invalid.

Description

The first call in a C+O program should be a call to TskInit or TskDefaultInit. Most, if not all, programs should initialize the pre-defined global TskMain. Once a task has been initialized, a call should be made to TskOnException. This sets up an exception handler for the program which will catch many program logic errors, especially calls to C+O.

Glossary and Special Terms

No special glossary exists for this class.

Class Implementation *Task*

Instance Variables

The following is the data structure definition for *Task*.

```
struct Task {        MediumInt      tskCheck;
                     MediumInt      argc;
                     PSTR           *argv;
                     Dpa            exhStack;
                     MediumInt      current;
                     PSTR           fileName;
                     MediumInt      lineNo;
                     MediumInt      condition;
                     ExceptionType    type;
                     ExcFilter      defExcFilter; }
```

tskCheck	Uniquely identifies *Task* instances.
argc	Argc from main().
*argv	Argv from main().
exhStack	Stack of exception handlers.
current	Active exception handler index.
fileName	File which invoked exception.
lineNo	Line number in file which invoked exception.
condition	Condition value at time of exception.
type	Type of exception.
defExcFilter	Filter for exception handler.

Superclasses

None

Messages and Responses

None

Class Variables

Type	Name	Description
Task	TskMain	The default task for the program

Function Listing In Alphabetic Order

Function Name	Description	Page	Scope	Macro
TskBasicAssert	Foundation function	N/A	Undoc	N
TskCondition	Raise exception conditionally	Tsk - 2	Public	N
TskDeInit	Deinitialize instance	Tsk - 4	Public	N
TskDefaultInit	Initialize instance with defaults	Tsk - 5	Public	N
TskExit	Exit program with code	Tsk - 6	Public	N
TskExitWithMsg	Exit program and print message	Tsk - 7	Public	N
TskGetArgc	Get main() argument count	Tsk - 8	Public	N
TskGetArgv	Get main() argument vector	Tsk - 9	Public	N
TskGetExceptionCondition	Return exception condition	Tsk - 10	Public	N
TskGetExceptionFileName	Return filename of exception	Tsk - 11	Public	N
TskGetExceptionLineNo	Return exception line number	Tsk - 12	Public	N
TskGetExceptionType	Return exception type	Tsk - 13	Public	N
TskInit	Initialize instance	Tsk - 15	Public	N
TskIsInitialized	Is the *Task* initialized	N/A	Undoc	N
TskLogCond	Raise exception conditionally	Tsk - 17	Public	N
TskMainLogCond	Raise exception conditionally	Tsk - 19	Public	N
TskMainPreCond	Raise exception conditionally	Tsk - 21	Public	N
TskMainPtrCond	Raise exception conditionally	Tsk - 23	Public	N
TskMainRaiseException	Raise exception unconditionally	Tsk - 25	Public	N
TskNormalExit	Exit task normally	Tsk - 27	Public	N
TskOnException	Establish exception handler	Tsk - 28	Public	N
TskPopExceptionHandler	Pop exception handler	Tsk - 30	Public	N
TskPreCond	Raise exception conditionally	Tsk - 32	Public	N
TskPrintException	Print description of exception	Tsk - 34	Public	N
TskPropagateException	Pass last exception	Tsk - 36	Public	N
TskPtrCond	Raise exception conditionally	Tsk - 38	Public	N
TskPushExh	Push the exception	N/A	Undoc	N
TskRaiseException	Raise exception unconditionally	Tsk - 40	Public	N

Function Listing By Scope

Public Functions

Function Name	Description	Page	Macro
TskCondition	Raise exception conditionally	Tsk - 2	N
TskDeInit	Deinitialize instance	Tsk - 4	N
TskDefaultInit	Initialize instance with defaults	Tsk - 5	N
TskExit	Exit program with code	Tsk - 6	N
TskExitWithMsg	Exit program and print message	Tsk - 7	N
TskGetArgc	Get main() argument count	Tsk - 8	N
TskGetArgv	Get main() argument vector	Tsk - 9	N
TskGetExceptionCondition	Return exception condition	Tsk - 10	N
TskGetExceptionFileName	Return filename of exception	Tsk - 11	N
TskGetExceptionLineNo	Return exception line number	Tsk - 12	N
TskGetExceptionType	Return exception type	Tsk - 13	N
TskInit	Initialize instance	Tsk - 15	N
TskLogCond	Raise exception conditionally	Tsk - 17	N
TskMainLogCond	Raise exception conditionally	Tsk - 19	N
TskMainPreCond	Raise exception conditionally	Tsk - 21	N
TskMainPtrCond	Raise exception conditionally	Tsk - 23	N
TskMainRaiseException	Raise exception unconditionally	Tsk - 25	N
TskNormalExit	Exit task normally	Tsk - 27	N
TskOnException	Establish exception handler	Tsk - 28	N
TskPopExceptionHandler	Pop exception handler	Tsk - 30	N
TskPreCond	Raise exception conditionally	Tsk - 32	N
TskPrintException	Print description of exception	Tsk - 34	N
TskPropagateException	Pass last exception	Tsk - 36	N
TskPtrCond	Raise exception conditionally	Tsk - 38	N
TskRaiseException	Raise exception unconditionally	Tsk - 40	N

Private Functions

None			

Undocumented Functions

Function Name	Description	Page	Macro
TskBasicAssert	Foundation function	N/A	N
TskIsInitialized	Is the *Task* initialized	N/A	N
TskPushExh	Push the exception	N/A	N

Friend Functi ns

None			

Listing Of Functions by Category

Undocumented Functions

Function Name	Description	Page	Scope	Macro
TskBasicAssert	Foundation function	N/A	Undoc	N
TskIsInitialized	Is the *Task* initialized	N/A	Undoc	N
TskPushExh	Push the exception	N/A	Undoc	N

Exception Functions

Function Name	Description	Page	Scope	Macro
TskCondition	Raise exception conditionally	Tsk - 2	Public	N
TskLogCond	Raise exception conditionally	Tsk - 17	Public	N
TskMainLogCond	Raise exception conditionally	Tsk - 19	Public	N
TskMainPreCond	Raise exception conditionally	Tsk - 21	Public	N
TskMainPtrCond	Raise exception conditionally	Tsk - 23	Public	N
TskMainRaiseException	Raise exception unconditionally	Tsk - 25	Public	N
TskOnException	Establish exception handler	Tsk - 28	Public	N
TskPopExceptionHandler	Pop exception handler	Tsk - 30	Public	N
TskPreCond	Raise exception conditionally	Tsk - 32	Public	N
TskPrintException	Print description of exception	Tsk - 34	Public	N
TskPropagateException	Pass last exception	Tsk - 36	Public	N
TskPtrCond	Raise exception conditionally	Tsk - 38	Public	N
TskRaiseException	Raise exception unconditionally	Tsk - 40	Public	N

Initialization Functions

Function Name	Description	Page	Scope	Macro
TskDeInit	Deinitialize instance	Tsk - 4	Public	N
TskDefaultInit	Initialize instance with defaults	Tsk - 5	Public	N
TskInit	Initialize instance	Tsk - 15	Public	N

Exit Functions

Function Name	Description	Page	Scope	Macro
TskExit	Exit program with code	Tsk - 6	Public	N
TskExitWithMsg	Exit program and print message	Tsk - 7	Public	N
TskNormalExit	Exit task normally	Tsk - 27	Public	N

Query Functions

Function Name	Description	Page	Scope	Macro
TskGetArgc	Get main() argument count	Tsk - 8	Public	N
TskGetArgv	Get main() argument vector	Tsk - 9	Public	N
TskGetExceptionCondition	Return exception condition	Tsk - 10	Public	N
TskGetExceptionFileName	Return filename of exception	Tsk - 11	Public	N
TskGetExceptionLineNo	Return exception line number	Tsk - 12	Public	N
TskGetExceptionType	Return exception type	Tsk - 13	Public	N

Listing Of Functions With Macros Available

None			

Class Description for *Vertex*

Structure Name: Vertex

Abbreviation: Vtx

Class Type: Inheritable class

Introduction

A *Vertex* must be used in combination with *Graph* and *Edge* to be of any value. A *Vertex* can model among other things, activities in a critical path network, cities connected by highways, cells in a spreadsheet, etc.

The *Vertex* class has two *friend* classes *Edge* and *Graph*. The documentation for the *Vertex* class should therefore be read in conjunction with the classes *Edge* and *Graph*.

Description

A vertex can have **incoming edges** and **outgoing edges**. In Figure 1 below the vertex A has an incoming edge x, and an outgoing edge y.

Glossary and Special Terms

The **successor vertices** of a *Vertex* A are the vertices at the head of the outgoing edges of A. The **predecessor vertices** of A are the vertices at the tail end of the incoming edges of A. See Figure 1.

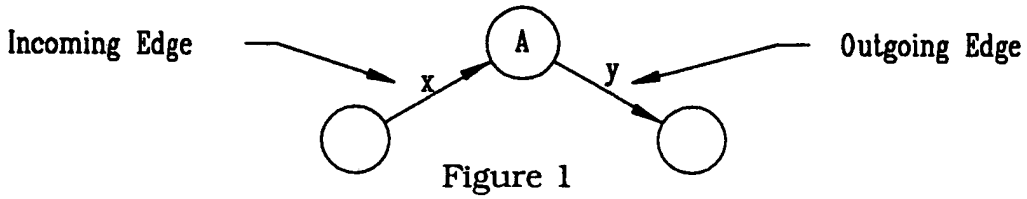


Figure 1

Class Implementation *Vertex*

Instance Variables

The following is the data structure definition for a vertex in a directed graph.

```
struct Vertex {     Obj          object;
                    MediumInt    vtxCheck;
                    LeI          graph;
                    DII          inList;
                    DII          outList; };
```

obj	Gives *Vertex* class object-oriented properties of inheritance and dynamic binding.
vtxCheck	Uniquely identifies *Vertex* instances.
graph	*List* element in a graph.
inList	*List* of incoming edges.
outList	*List* of outgoing edges.

Superclasses

Type	Name	Description
ListElement	GrfLeI	*Graph* membership
List	InDII	*List* of incoming edges
List	OutDII	*List* of outgoing edges

Messages and Responses

Vertex overrides the destroy message from both of its *List* superclasses and its *ListElement* superclass. The implementation of the destroy message is handled by VtxDestroy. The subclass of *Vertex* should override this method.

Selector	Method	Description
destroy	VtxDestroy	Destroy the object

Class Variables

Type	Name	Description
Mcl	VtxMcl	Metaclass decription for *Vertex*.
MediumInt	VtxInDllOffset	Client offset of *Vertex* from superclass InDll.
MediumInt	VtxOutDllOffset	Client offset of *Vertex* from superclass OutDll.
MediumInt	VtxGrfLelOffset	Client offset of *Vertex* from superclass GrfLel.

Function Listing In Alphabetic Order

Function Name	Description	Page	Scope	Macro
VtxAsGrfLel	Return list element in graph	Vtx - 2	Friend	Y
VtxAsInDll	Return vertex as list of incoming edges	Vtx - 3	Friend	N
VtxAsObj	Return edge as object	Vtx - 4	Private	N
VtxAsOutDll	Return vertex as list of outgoing edges	Vtx - 5	Friend	N
VtxClear	Clear vertex	Vtx - 6	Public	N
VtxConnectToGrf	Connect vertex to graph	Vtx - 7	Public	N
VtxCountIn	Count incoming edges	Vtx - 9	Public	Y
VtxCountOut	Count outgoing edges	Vtx - 11	Public	Y
VtxDeInit	Deinitialize the *Vertex* object	Vtx - 13	Public	N
VtxDestroy	Deinitialize *Vertex* object and free space	Vtx - 14	Public	N
VtxDisconnectFromGrf	Disconnect vertex from graph	Vtx - 15	Public	Y
VtxFindOutEdg	Visit search function: outgoing edges	Vtx - 17	Private	N
VtxFindOutEdgClient	Visit search function: outgoing edges	Vtx - 19	Public	N
VtxGetClient	Return client of vertex	Vtx - 21	Public	Y
VtxGetFirstIn	Return first incoming edge	Vtx - 22	Public	Y
VtxGetFirstOut	Return first outgoing edge	Vtx - 23	Public	Y
VtxGetGrf	Return graph	Vtx - 24	Friend	Y
VtxInGrf	Is vertex in graph	Vtx - 26	Public	Y
VtxInit	Initialize the *Vertex* object	Vtx - 25	Public	N
VtxSendDestroy	Send message for vertex destruction	Vtx - 27	Public	N
VtxStackSetup	Set up values for topsort	Vtx - 28	Friend	Y
VtxVisitEdge	Visit function: each edge	Vtx - 29	Friend	N
VtxVisitEdgeClient	Visit function: each edge	Vtx - 31	Public	N
VtxVisitInEdge	Visit function: incoming edge	Vtx - 33	Friend	N
VtxVisitInEdgeClient	Visit function: incoming edge	Vtx - 35	Public	N
VtxVisitOutEdge	Visit function: outgoing edge	Vtx - 37	Friend	N
VtxVisitOutEdgeClient	Visit function: outgoing edge	Vtx - 39	Public	N

Function Listing By Scope

Public Functions

Function Name	Description	Page	Macro
VtxClear	Clear vertex	Vtx - 6	N
VtxConnectToGrf	Connect vertex to graph	Vtx - 7	N
VtxCountIn	Count incoming edges	Vtx - 9	N
VtxCountOut	Count outgoing edges	Vtx - 11	N
VtxDeInit	Deinitialize the *Vertex* object	Vtx - 13	N
VtxDestroy	Deinitialize *Vertex* object and free space	Vtx - 14	N
VtxDisconnectFromGrf	Disconnect vertex from graph	Vtx - 15	N
VtxFindOutEdgClient	Visit search function: outgoing edges	Vtx - 19	N
VtxGetClient	Return client of vertex	Vtx - 21	N
VtxGetFirstIn	Return first incoming edge	Vtx - 22	N
VtxGetFirstOut	Return first outgoing edge	Vtx - 23	N
VtxInGrf	Is vertex in graph	Vtx - 26	N
VtxInit	Initialize the *Vertex* object	Vtx - 25	N
VtxSendDestroy	Send message for vertex destruction	Vtx - 27	N
VtxVisitEdgeClient	Visit function: each edge	Vtx - 31	N
VtxVisitInEdgeClient	Visit function: incoming edge	Vtx - 35	N
VtxVisitOutEdgeClient	Visit function: outgoing edge	Vtx - 39	N

Private Functions

Function Name	Description	Page	Macro
VtxAsObj	Return edge as object	Vtx - 4	N
VtxFindOutEdg	Visit search function: outgoing edges	Vtx - 17	N

Undocumented Functions

None			

Friend Functions

Function Name	Description	Page	Macro
VtxAsGrfLel	Return list element in graph	Vtx - 2	N
VtxAsInDll	Return vertex as list of incoming edges	Vtx - 3	N
VtxAsOutDll	Return vertex as list of outgoing edges	Vtx - 5	N
VtxGetGrf	Return graph	Vtx - 24	N
VtxStackSetup	Set up values for topsort	Vtx - 28	N
VtxVisitEdge	Visit function: each edge	Vtx - 29	N
VtxVisitInEdge	Visit function: incoming edge	Vtx - 33	N
VtxVisitOutEdge	Visit function: outgoing edge	Vtx - 37	N

Listing Of Functions by Category

Superclass Functions

Function Name	Description	Page	Scope	Macro
VtxAsGrfLel	Return list element in graph	Vtx - 2	Friend	Y
VtxAsInDll	Return vertex as list of incoming edges	Vtx - 3	Friend	N
VtxAsObj	Return edge as object	Vtx - 4	Private	N
VtxAsOutDll	Return vertex as list of outgoing edges	Vtx - 5	Friend	N

Edit Functions

Function Name	Description	Page	Scope	Macro
VtxClear	Clear vertex	Vtx - 6	Public	N
VtxConnectToGrf	Connect vertex to graph	Vtx - 7	Public	N
VtxDisconnectFromGrf	Disconnect vertex from graph	Vtx - 15	Public	Y

Query Functions

Function Name	Description	Page	Scope	Macro
VtxCountIn	Count incoming edges	Vtx - 9	Public	Y
VtxCountOut	Count outgoing edges	Vtx - 11	Public	Y
VtxFindOutEdg	Visit search function: outgoing edges	Vtx - 17	Private	N
VtxFindOutEdgClient	Visit search function: outgoing edges	Vtx - 19	Public	N
VtxGetClient	Return client of vertex	Vtx - 21	Public	Y
VtxGetFirstIn	Return first incoming edge	Vtx - 22	Public	Y
VtxGetFirstOut	Return first outgoing edge	Vtx - 23	Public	Y
VtxGetGrf	Return graph	Vtx - 24	Friend	Y
VtxInGrf	Is vertex in graph	Vtx - 26	Public	Y
VtxStackSetup	Set up values for topsort	Vtx - 28	Friend	Y

Initialization Functions

Function Name	Description	Page	Scope	Macro
VtxDeInit	Deinitialize the *Vertex* object	Vtx - 13	Public	N
VtxDestroy	Deinitialize *Vertex* object and free space	Vtx - 14	Public	N
VtxInit	Initialize the *Vertex* object	Vtx - 25	Public	N
VtxSendDestroy	Send message for vertex destruction	Vtx - 27	Public	N

Visit Functions

Function Name	Description	Page	Scope	Macro
VtxVisitEdge	Visit function: each edge	Vtx - 29	Friend	N
VtxVisitEdgeClient	Visit function: each edge	Vtx - 31	Public	N
VtxVisitInEdge	Visit function: incoming edge	Vtx - 33	Friend	N
VtxVisitInEdgeClient	Visit function: incoming edge	Vtx - 35	Public	N
VtxVisitOutEdge	Visit function: outgoing edge	Vtx - 37	Friend	N
VtxVisitOutEdgeClient	Visit function: outgoing edge	Vtx - 39	Public	N

Listing Of Functions With Macros Available

Function Name	Description	Page	Scope
VtxAsGrfLel	Return list element in graph	Vtx - 2	Friend
VtxCountIn	Count incoming edges	Vtx - 9	Public
VtxCountOut	Count outgoing edges	Vtx - 11	Public
VtxDisconnectFromGrf	Disconnect vertex from graph	Vtx - 15	Public
VtxGetClient	Return client of vertex	Vtx - 21	Public
VtxGetFirstIn	Return first incoming edge	Vtx - 22	Public
VtxGetFirstOut	Return first outgoing edge	Vtx - 23	Public
VtxGetGrf	Return graph	Vtx - 24	Friend
VtxInGrf	Is vertex in graph	Vtx - 26	Public
VtxStackSetup	Set up values for topsort	Vtx - 28	Friend

This page is intentionally left blank.

Class Exceptions Reference

BlkClear(pBlk)

BLK - 65:	pBlk must be a valid Blk pointer.

BlkDeInit(pBlk)

BLK - 89:	pBlk must be a valid Blk pointer.
BLK - 91:	pBlk must be a valid Blk pointer.

BlkExecute(pBlk, pObj)

BLK - 115:	pBlk must be a valid Blk pointer.
BLK - 116:	pBlk must contain a non Null method pointer.

BlkExecuteRetBool(pBlk, pObj)

BLK - 156:	pBlk must be a valid Blk pointer.
BLK - 157:	pBlk must contain a non Null method pointer.

BlkExecuteRetDataPtr(pBlk, pObj)

BLK - 197:	pBlk must be a valid Blk pointer.
BLK - 198:	pBlk must contain a non Null method pointer.

BlkExecuteRetFuncPtr(pBlk, pObj)

BLK - 238:	pBlk must be a valid Blk pointer.
BLK - 239:	pBlk must contain a non Null method pointer.

BlkExecuteRetInt(pBlk, pObj)

BLK - 279:	pBlk must be a valid Blk pointer.
BLK - 280:	pBlk must contain a non Null method pointer.

Class Exceptions Reference

BlkHasMethod(pBlk)

BLK - 317:	pBlk must be a valid Blk pointer.

BlkInit(pBlk, pMth)

BLK - 343:	pBlk must be a valid Blk pointer.

BlkPrint(pBlk, pCio, item, level, name)

BLK - 380:	pBlk must be a valid Blk pointer.
BLK - 381:	pCio must be a valid Cio pointer.

BlkPushDataPtr(pBlk, p)

BLK - 410:	pBlk must be a valid Blk pointer.
BLK - 411:	The total number of parameters must not be larger than the parameter array.

BlkPushFuncPtr(pBlk, p)

BLK - 441:	pBlk must be a valid Blk pointer.

BlkPushLargeInt(pBlk, p)

BLK - 470:	pBlk must be a valid Blk pointer.
BLK - 471:	The total number of parameters must not be larger than the parameter array.

BlkPushMediumInt(pBlk, p)

BLK - 497:	pBlk must be a valid Blk pointer.
BLK - 498:	The total number of parameters must not be larger than the parameter array.

BlkSetMethod(pBlk, pMth)

BLK - 524:	pBlk must be a valid Blk pointer.
BLK - 525:	pMth must be a valid Blk pointer.

ClsCreateMessages(pCls)

CLS - 97:	pCls must be a valid Cls pointer.

ClsCreateObject(pCls)

CLS - 141:	pCls must be a valid Cls pointer.
CLS - 142:	The class must not have any sub-classes.
CLS - 145:	Not enough memory to create object

ClsCreateObjectFast(pCls)

CLS - 197:	pCls must be a valid Cls pointer.
CLS - 198:	The class must not have any sub-classes.
CLS - 201:	Not enough memory to create object

ClsCreateSupers(pCls)

CLS - 233:	pCls must be a valid Cls pointer.

ClsDeInit(pCls)

CLS - 268:	pCls must be a valid Cls pointer.
CLS - 272:	pCls must be a valid Cls pointer.

ClsDestroy(pCls)

CLS - 305:	pCls must be a valid Cls pointer.

ClsDestroyMessages(pCls)

CLS - 334:	pCls must be a valid Cls pointer.

ClsDestroyObject(pCls, pObj)

CLS - 370:	pCls must be a valid Cls pointer.
CLS - 371:	pObj must be a valid Obj pointer.
CLS - 372:	The class must not have any sub-classes.

Class Exceptions Reference

ClsDestroySuperClasses(pCls)

CLS - 402:	pCls must be a valid Cls pointer.

ClsFindFirstMsg(pCls, pStr)

CLS - 446:	pCls must be a valid Cls pointer.

ClsFindFirstSelectorIndex(pCls, pStr)

CLS - 490:	pCls must be a valid Cls pointer.

ClsFindMsg(pCls, pStr)

CLS - 529:	pCls must be a valid Cls pointer.

ClsFindSelectorIndex(pCls, pStr)

CLS - 572:	pCls must be a valid Cls pointer.

ClsFindSuperClass(pCls, pStr)

CLS - 609:	pCls must be a valid Cls pointer.

ClsGetMessageCount(pCls)

CLS - 641:	pCls must be a valid Cls pointer.

ClsGetMethodAndOffset(pCls, m, ppMth, pOffset)

CLS - 692:	pCls must be a valid Cls pointer.
CLS - 693:	The message number must be greater than or equal to zero, and less than the number of messages.

ClsGetName(pCls)

CLS - 721:	pCls must be a valid Cls pointer.

ClsGetNthMsg(pCls, n)

CLS - 755:	pCls must be a valid Cls pointer.
CLS - 756:	The message number must be greater than or equal to zero, and less than the number of messages.

ClsGetNthSuperClass(pCls, n)

CLS - 788:	pCls must be a valid Cls pointer.
CLS - 789:	The super class index must be greater than or equal to zero, and less than the number of superclasses.

ClsGetObjectSize(pCls)

CLS - 821:	pCls must be a valid Cls pointer.

ClsGetOffsetForMsg(pCls, m)

CLS - 861:	pCls must be a valid Cls pointer.
CLS - 862:	The message number must be greater than or equal to zero, and less than the number of messages.

ClsGetOffsetOfNthSuper(pCls, n)

CLS - 899:	pCls must be a valid Cls pointer.
CLS - 900:	The super class index must be greater than or equal to zero, and less than the number of superclasses.

ClsGetRootSubClass(pCls)

CLS - 932:	pCls must be a valid Cls pointer.

ClsGetRootSubObjectOffset(pCls)

CLS - 966:	pCls must be a valid Cls pointer.

ClsGetRootSubObjectSize(pCls)

CLS - 998:	pCls must be a valid Cls pointer.

Class Exceptions Reference

ClsGetSize(pCls)

CLS - 1029:	pCls must be a valid Cls pointer.

ClsGetSubClass(pCls)

CLS - 1061:	pCls must be a valid Cls pointer.

ClsGetSubObjectOffset(pCls)

CLS - 1092:	pCls must be a valid Cls pointer.

ClsGetSuperClassCount(pCls)

CLS - 1123:	pCls must be a valid Cls pointer.

ClsGetSuperClassIndex(pCls)

CLS - 1152:	pCls must be a valid Cls pointer.

ClsInit(pCls, pMcl)

CLS - 1183:	pMcl must be a valid Mcl pointer.
CLS - 1184:	pCls must be a valid Cls pointer.
CLS - 1187:	Not enough memory to allocate superclasses
CLS - 1189:	Not enough memory to allocate messages

ClsIsRoot(pCls)

CLS - 1223:	pCls must be a valid Cls pointer.

ClsOffsetSupers(pClsSub, pClsSuper, offset)

CLS - 1252:	pClsSub must be a valid Cls pointer.
CLS - 1253:	pClsSuper must be a valid Cls pointer.

ClsPrint(pCls, pCio, item, level, name)

CLS - 1293:	pCls must be a valid Cls pointer.
CLS - 1294:	pCio must be a valid Cio pointer.

ClsSendDestroy(pCls)

CLS - 1327:	pCls must be a valid Cls pointer.

ClsSendObjectMessage(pCls, pObj, m, pBlk)

CLS - 1373:	pCls must be a valid Cls pointer.
CLS - 1374:	pObj must be a valid Obj pointer.
CLS - 1375:	The message number must be greater than or equal to zero, and less than the number of messages.
CLS - 1376:	pBlk must be a valid Blk pointer.

ClsSendObjectMessageReturnInt(pCls, pObj, m, pBlk)

CLS - 1426:	pCls must be a valid Cls pointer.
CLS - 1427:	pObj must be a valid Obj pointer.
CLS - 1428:	The message number must be greater than or equal to zero, and less than the number of messages.
CLS - 1429:	pBlk must be a valid Blk pointer.

ClsSendObjectMessageReturnPtr(pCls, pObj, m, pBlk)

CLS - 1479:	pCls must be a valid Cls pointer.
CLS - 1480:	pObj must be a valid Obj pointer.
CLS - 1481:	The message number must be greater than or equal to zero, and less than the number of messages.
CLS - 1482:	pBlk must be a valid Blk pointer.

ClsSuperClassOf(pClsSub, pClsSuper, index, offset)

CLS - 1511:	pClsSub must be a valid Cls pointer.
CLS - 1512:	pClsSuper must be a valid Cls pointer.
CLS - 1513:	pClsSub must not be identical to pClsSuper

DllAppend(pDll, pLel, pLelApp)

DLL - 83:	pDll must be a valid Dll pointer.
DLL - 84:	pLelApp must be a valid Lel pointer.

DllAppendLast(pDll, pLel)

DLL - 118:	pDll must be a valid Dll pointer.
DLL - 119:	pLel must be a valid Lel pointer.

DllAsObj(pDll)

DLL - 149:	pDll must be a valid Dll pointer.

DllClear(pDll)

DLL - 175:	pDll must be a valid Dll pointer.

DllCut(pDll, pLel)

DLL - 208:	pDll must be a valid Dll pointer.
DLL - 209:	pLel must be a valid Lel pointer.
DLL - 210:	pLel must be in the list pDll.

DllCutChildren(pDll)

DLL - 236:	pDll must be a valid Dll pointer.

DllCutRange(pDll, pLelBeg, pLelEnd)

DLL - 282:	pDll must be a valid Dll pointer.
DLL - 283:	pLelBeg must be a valid Lel pointer.
DLL - 284:	pLelEnd must be a valid Lel pointer.
DLL - 285:	If pLelBeg does not equal pLelEnd then pLelBeg must precede pLelEnd.
DLL - 286:	pLelBeg must be in the list pDll.

DllDeInit(pDll)

DLL - 316:	pDll must be a valid Dll pointer.
DLL - 317:	pDll cannot have any list elements
DLL - 319:	pDll must be a valid Dll pointer.

DllDestroy(pDll)

DLL - 348:	pDll must be a valid Dll pointer.
DLL - 349:	pDll must not have a sub-object

DllGetClient(pDll, offset)

DLL - 384:	pDll must be a valid Dll pointer.

DllGetFirst(pDll)

DLL - 415:	pDll must be a valid Dll pointer.

DllGetLast(pDll)

DLL - 447:	pDll must be a valid Dll pointer.

DllGetNth(pDll, index)

DLL - 484:	pDll must be a valid Dll pointer.

DllGetSize(pDll)

DLL - 512:	pDll must be a valid Dll pointer.

DllInit(pDll)

DLL - 540:	pDll must be a valid Dll pointer.

DllInsert(pDll, pLel, pLelIns)

DLL - 581:	pDll must be a valid Dll pointer.
DLL - 582:	pLelIns must be a valid Lel pointer.

DllInsertFirst(pDll, pLel)

DLL - 615:	pDll must be a valid Dll pointer.
DLL - 616:	pLel must be a valid Lel pointer.

DllIsEmpty(pDll)

DLL - 644:	pDll must be a valid Dll pointer.

Class Exceptions Reference

DllLelClientCount(pDll, offset, pBlk)

DLL - 691:	pDll must be a valid Dll pointer.
DLL - 692:	pBlk must be a valid Blk pointer.

DllLelClientFind(pDll, offset, pBlk)

DLL - 739:	pDll must be a valid Dll pointer.
DLL - 740:	pBlk must be a valid Blk pointer.

DllLelClientFirst(pDll, offset)

DLL - 776:	pDll must be a valid Dll pointer.

DllLelClientGetNth(pDll, offset, index)

DLL - 819:	pDll must be a valid Dll pointer.

DllLelClientLast(pDll, offset)

DLL - 856:	pDll must be a valid Dll pointer.

DllLelClientVisitBwd(pDll, offset, pBlk)

DLL - 901:	pDll must be a valid Dll pointer.
DLL - 902:	pBlk must be a valid Blk pointer.

DllLelClientVisitFwd(pDll, offset, pBlk)

DLL - 946:	pDll must be a valid Dll pointer.
DLL - 947:	pBlk must be a valid Blk pointer.

DllNotifyCutRange(pDll, pLelBeg, pLelEnd)

DLL - 974:	pDll must be a valid Dll pointer.
DLL - 975:	pLelBeg must be a valid Lel pointer.
DLL - 976:	pLelEnd must be a valid Lel pointer.

DllNotifyPasteRange(pDll, pLelBeg, pLelEnd)

DLL - 1005:	pDll must be a valid Dll pointer.
DLL - 1006:	pLelBeg must be a valid Lel pointer.
DLL - 1007:	pLelEnd must be a valid Lel pointer.

DllPasteRangeAfter(pDll, pLel, pLelBeg, pLelEnd)

DLL - 1063:	pDll must be a valid Dll pointer.
DLL - 1064:	pLelBeg must be a valid Lel pointer.
DLL - 1065:	pLelEnd must be a valid Lel pointer.
DLL - 1066:	If pLelBeg does not equal pLelEnd then pLelBeg must precede pLelEnd.
DLL - 1067:	pLelBeg must not already be in a list.
DLL - 1072:	pLel must be a valid Lel pointer.
DLL - 1073:	pLel must be in the list pDll.

DllPasteRangeBefore(pDll, pLel, pLelBeg, pLelEnd)

DLL - 1128:	pDll must be a valid Dll pointer.
DLL - 1129:	pLelBeg must be a valid Lel pointer.
DLL - 1130:	pLelEnd must be a valid Lel pointer.
DLL - 1131:	If pLelBeg does not equal pLelEnd then pLelBeg must precede pLelEnd.
DLL - 1132:	pLelBeg must not already be in a list.
DLL - 1137:	pLel must be a valid Lel pointer.
DLL - 1138:	pLel must be in the list pDll.

DllPasteRangeFirst(pDll, pLelBeg, pLelEnd)

DLL - 1195:	pDll must be a valid Dll pointer.
DLL - 1196:	pLelBeg must be a valid Lel pointer.
DLL - 1197:	pLelEnd must be a valid Lel pointer.

DllPasteRangeLast(pDll, pLelBeg, pLelEnd)

DLL - 1253:	pDll must be a valid Dll pointer.
DLL - 1254:	pLelBeg must be a valid Lel pointer.
DLL - 1255:	pLelEnd must be a valid Lel pointer.

DllSendDestroy(pDll)

DLL - 1282:	pDll must be a valid Dll pointer.

Class Exceptions Reference

DpaAppend(pDpa, pObj)

DPA - 96:	pDpa must be a valid Dpa pointer.

DpaClear(pDpa)

DPA - 122:	pDpa must be a valid Dpa pointer.

DpaCount(pDpa, pBlk)

DPA - 171:	pDpa must be a valid Dpa pointer.
DPA - 172:	pBlk must be a valid Blk pointer.

DpaCountRange(pDpa, beg, end, pBlk)

DPA - 233:	pDpa must be a valid Dpa pointer.
DPA - 234:	The index beg must be in the range [0-END].
DPA - 235:	The index end must be in the range [0-END].
DPA - 236:	beg must be less than or equal to end.
DPA - 237:	pBlk must be a valid Blk pointer.

DpaDeInit(pDpa)

DPA - 271:	pDpa must be a valid Dpa pointer.
DPA - 274:	pDpa must be a valid Dpa pointer.

DpaDelete(pDpa, beg, n)

DPA - 316:	pDpa must be a valid Dpa pointer.
DPA - 317:	The index beg must be in the range [0-END].
DPA - 318:	(beg + n) must be less than or equal to the size of the array.

DpaDestroy(pDpa)

DPA - 346:	pDpa must be a valid Dpa pointer.

DpaExpand(pDpa, beg, n)

DPA - 397:	pDpa must be a valid Dpa pointer.
DPA - 398:	beg must be in the range [0:DpaGetSize]
DPA - 399:	n must be greater than zero

DpaFind(pDpa, pBlk)

DPA - 448:	pDpa must be a valid Dpa pointer.
DPA - 449:	pBlk must be a valid Blk pointer.

DpaFindPtrBwd(pDpa, beg, pObj)

DPA - 493:	pDpa must be a valid Dpa pointer.
DPA - 494:	The index beg must be in the range [0-END].

DpaFindPtrFwd(pDpa, beg, pObj)

DPA - 539:	pDpa must be a valid Dpa pointer.
DPA - 540:	The index beg must be in the range [0-END].

DpaFindRangeBwd(pDpa, beg, end, pBlk)

DPA - 601:	pDpa must be a valid Dpa pointer.
DPA - 602:	The index beg must be in the range [0-END].
DPA - 603:	The index end must be in the range [0-END].
DPA - 604:	beg must be less than or equal to end.

DpaFindRangeFwd(pDpa, beg, end, pBlk)

DPA - 664:	pDpa must be a valid Dpa pointer.
DPA - 665:	The index beg must be in the range [0-END].
DPA - 666:	The index end must be in the range [0-END].
DPA - 667:	beg must be less than or equal to end.

DpaGetNth(pDpa, n)

DPA - 706:	pDpa must be a valid Dpa pointer.
DPA - 707:	The index n must be in the range [0-END].

DpaGetLast(pDpa)

DPA - 737:	pDpa must be a valid Dpa pointer.
DPA - 738:	The array must have at least one element

Class Exceptions Reference

DpaGetSize(pDpa)

DPA - 769:	pDpa must be a valid Dpa pointer.

DpaInit(pDpa, s, i)

DPA - 806:	pDpa cannot be NULL
DPA - 807:	The increment must be greater than one
DPA - 808:	The initial size must be greater than or equal to zero
DPA - 810:	pDpa must be a valid Dpa pointer.

DpaNewArray(pDpa)

DPA - 837:	pDpa must be a valid Dpa pointer.
DPA - 846:	The allocated size is not a multiple of the increment
DPA - 850:	Not enough memory to allocate array

DpaResize(pDpa, n)

DPA - 874:	pDpa must be a valid Dpa pointer.
DPA - 875:	N must be greater than or equal to zero

DpaSetNth(pDpa, index, pObj)

DPA - 915:	pDpa must be a valid Dpa pointer.
DPA - 916:	The index index must be in the range [0-END].

DpaSetRegionNull(pDpa, beg, n)

DPA - 958:	pDpa must be a valid Dpa pointer.
DPA - 959:	The index beg must be in the range [0-END].
DPA - 960:	(beg + n) must be less than or equal to the size of the array.

DpaSetSize(pDpa, size)

DPA - 995:	pDpa must be a valid Dpa pointer.
DPA - 996:	The new size must be greater than or equal to zero

DpaShiftDown(pDpa, n)

DPA - 1043:	pDpa must be a valid Dpa pointer.
DPA - 1044:	N must be greater than zero and less than the aray size

DpaShiftUp(pDpa, n)

DPA - 1075:	pDpa must be a valid Dpa pointer.
DPA - 1076:	N must be greater than zero and less than the array size

DpaVisit(pDpa, pBlk)

DPA - 1114:	pDpa must be a valid Dpa pointer.
DPA - 1115:	pBlk must be a valid Blk pointer.

DpaVisitClient(pDpa, offset, pBlk)

DPA - 1163:	pDpa must be a valid Dpa pointer.
DPA - 1164:	pBlk must be a valid Blk pointer.

DpaVisitRange(pDpa, beg, end, pBlk)

DPA - 1216:	pDpa must be a valid Dpa pointer.
DPA - 1217:	The index beg must be in the range [0-END].
DPA - 1218:	The index end must be in the range [0-END].
DPA - 1219:	beg must be less than or equal to end.

DpaVisitRegion(pDpa, beg, n, pBlk)

DPA - 1272:	pDpa must be a valid Dpa pointer.
DPA - 1273:	The index beg must be in the range [0-END].
DPA - 1274:	(beg + n) must be less than or equal to the size of the array.

DpaVisitSelfAndSuccessors(pDpa, beg, pBlk)

DPA - 1318:	pDpa must be a valid Dpa pointer.
DPA - 1319:	The index beg must be in the range [0-END].
DPA - 1320:	pBlk must be a valid Blk pointer.

EdgAsGrfLel(pEdg)

EDG - 101:	pEdg must be a valid Edg pointer.

EdgAsInLel(pEdg)

EDG - 134:	pEdg must be a valid Edg pointer.

EdgAsObj(pEdg)

EDG - 166:	pEdg must be a valid Edg pointer.

EdgAsOutLel(pEdg)

EDG - 199:	pEdg must be a valid Edg pointer.

EdgClear(pEdg)

EDG - 223:	pEdg must be a valid Edg pointer.
EDG - 230:	Failed to disconnect all vertices

EdgCompareInVtx(pEdg, pVtx, offset, pBlk)

EDG - 289:	pEdg must be a valid Edg pointer.
EDG - 290:	pVtx must be a valid Vtx pointer.
EDG - 291:	pBlk must be a valid Blk pointer.

EdgConnectToGrf(pEdg, pGrf)

EDG - 327:	pEdg must be a valid Edg pointer.
EDG - 328:	pGrf must be a valid Grf pointer.
EDG - 329:	pEdg must not already be linked to a graph.
EDG - 330:	pEdg must not be linked to any vertices.

EdgConnectToVertices(pEdg, pVtxI, pVtxO)

EDG - 367:	pEdg must be a valid Edg pointer.
EDG - 368:	pVtxI must be a valid Vtx pointer.
EDG - 369:	pVtxO must be a valid Vtx pointer.
EDG - 370:	pEdg must be linked to a graph.
EDG - 371:	The Vertex pVtxI must be linked to a graph.
EDG - 372:	The Vertex pVtxO must be linked to a graph.
EDG - 373:	pEdg must not be linked to any vertices.

EdgDeInit(pEdg)

EDG - 402:	pEdg must be a valid Edg pointer.
EDG - 403:	pEdg must not already be linked to a graph.
EDG - 404:	pEdg must not be linked to any vertices.
EDG - 410:	pEdg must be a valid Edg pointer.

EdgDestroy(pEdg)

EDG - 438:	pEdg must be a valid Edg pointer.
EDG - 439:	pEdg must not have a sub-object

EdgDisconnectFromGrf(pEdg)

EDG - 469:	pEdg must be a valid Edg pointer.
EDG - 470:	pEdg must be linked to a graph.
EDG - 471:	pEdg must not be linked to any vertices.

EdgDisconnectFromVertices(pEdg)

EDG - 499:	pEdg must be a valid Edg pointer.
EDG - 500:	pEdg must be linked to a graph.
EDG - 506:	pEdg must not be linked to any vertices.

EdgGetClient(pEdg, offset)

EDG - 538:	pEdg must be a valid Edg pointer.

EdgGetGrf(pEdg)

EDG - 569:	pEdg must be a valid Edg pointer.

EdgGetInVtx(pEdg)

EDG - 605:	pEdg must be a valid Edg pointer.
EDG - 609:	pEdg points to an invalid Vertex

EdgGetNextIn(pEdg)

EDG - 642:	pEdg must be a valid Edg pointer.

EdgGetNextOut(pEdg)

EDG - 677:	pEdg must be a valid Edg pointer.

EdgGetOutVtx(pEdg)

EDG - 713:	pEdg must be a valid Edg pointer.
EDG - 717:	pEdg points to an invalid Vertex

EdgGetVertices(pEdg, ppVtxI, ppVtxO)

EDG - 760:	pEdg must be a valid Edg pointer.

EdgHasVertices(pEdg)

EDG - 790:	pEdg must be a valid Edg pointer.

EdgInGrf(pEdg)

EDG - 821:	pEdg must be a valid Edg pointer.

EdgInit(pEdg)

EDG - 850:	pEdg must be a valid Edg pointer.

EdgSendDestroy(pEdg)

EDG - 880:	pEdg must be a valid Edg pointer.

EdgUpdateInVtx(pEdg, pVtxI)

EDG - 912:	pEdg must be a valid Edg pointer.
EDG - 913:	pEdg must be linked to a graph.
EDG - 914:	pEdg must be linked to two vertices.
EDG - 915:	The Vertex pVtxI must be linked to a graph.

EdgUpdateOutVtx(pEdg, pVtxO)

EDG - 946:	pEdg must be a valid Edg pointer.
EDG - 947:	pEdg must be linked to a graph.
EDG - 948:	pEdg must be linked to two vertices.
EDG - 949:	The Vertex pVtxO must be linked to a graph.

GrfAnyCycles(pGrf)

GRF - 100:	pGrf must be a valid Grf pointer.

GrfAsEdgDll(pGrf)

GRF - 132:	pGrf must be a valid Grf pointer.

GrfAsObj(pGrf)

GRF - 165:	pGrf must be a valid Grf pointer.

GrfAsVtxDll(pGrf)

GRF - 197:	pGrf must be a valid Grf pointer.

GrfBasicTopologicalSort(pGrf, keepSort, fwdOnly)

GRF - 251:	pGrf must be a valid Grf pointer.

GrfClear(pGrf)

GRF - 414:	pGrf must be a valid Grf pointer.
GRF - 419:	Failed to disconnect all vertices
GRF - 420:	Failed to disconnect all edges

GrfCountEdg(pGrf)

GRF - 451:	pGrf must be a valid Grf pointer.

GrfCountVtx(pGrf)

GRF - 481:	pGrf must be a valid Grf pointer.

GrfDeInit(pGrf)

GRF - 513:	pGrf must be a valid Grf pointer.
GRF - 514:	pGrf must not contain any vertices
GRF - 515:	pGrf must not contain any edges
GRF - 523:	pGrf must be a valid Grf pointer.

GrfDestroy(pGrf)

GRF - 551:	pGrf must be a valid Grf pointer.
GRF - 552:	pGrf must not have a sub-object

GrfDoTopologicalSort(pGrf)

GRF - 588:	pGrf must be a valid Grf pointer.

GrfFindEdgClient(pGrf, offset, pBlk)

GRF - 640:	pGrf must be a valid Grf pointer.
GRF - 641:	pBlk must be a valid Blk pointer.

GrfFindVtxClient(pGrf, offset, pBlk)

GRF - 693:	pGrf must be a valid Grf pointer.
GRF - 694:	pBlk must be a valid Blk pointer.

GrfGetClient(pGrf, offset)

GRF - 728:	pGrf must be a valid Grf pointer.

GrfInit(pGrf)

GRF - 757:	pGrf must be a valid Grf pointer.

GrfSendDestroy(pGrf)

GRF - 789:	pGrf must be a valid Grf pointer.

GrfVisitEdgClient(pGrf, offset, pBlk)

GRF - 834:	pGrf must be a valid Grf pointer.
GRF - 835:	pBlk must be a valid Blk pointer.

GrfVisitVtxClient(pGrf, offset, pBlk)

GRF - 879:	pGrf must be a valid Grf pointer.
GRF - 880:	pBlk must be a valid Blk pointer.

GrfVisitVtxClientInTopOrderBwd(pGrf, offset, pBlk)

GRF - 927:	pGrf must be a valid Grf pointer.
GRF - 928:	pBlk must be a valid Blk pointer.

GrfVisitVtxClientInTopOrderFwd(pGrf, offset, pBlk)

GRF - 975:	pGrf must be a valid Grf pointer.
GRF - 976:	pBlk must be a valid Blk pointer.

JulAddDays(pJul, dy)

JUL - 68:	pJul must be a valid Jul pointer.

JulAddDaysL(pJul, dy)

JUL - 100:	pJul must be a valid Jul pointer.

JulAddMonths(pJul, mnth)

JUL - 138:	pJul must be a valid Jul pointer.

JulAddQuarters(pJul, qtrs)

JUL - 208:	pJul must be a valid Jul pointer.

JulAddYears(pJul, yr)

JUL - 279:	pJul must be a valid Jul pointer.

Class Exceptions Reference

JulCalendarToJulian(pJul, year, month, day)

JUL - 321:	pJul must be a valid Jul pointer.
JUL - 322:	Year must be in the range [1582:4713]
JUL - 323:	Month must be in the range [1:12]
JUL - 324:	Day must be a valid day for the month and year specified

JulCopy(pJulD, pJulS)

JUL - 364:	pJulD must be a valid Jul pointer.
JUL - 365:	pJulS must be a valid Jul pointer.

JulDateStrToJulian(pJul, pStr, format)

JUL - 411:	pJul must be a valid Jul pointer.

JulDayOfWeek(pJul)

JUL - 454:	pJul must be a valid Jul pointer.

JulDayOfYear(pJul)

JUL - 489:	pJul must be a valid Jul pointer.

JulDaysInMonth(pJul)

JUL - 532:	pJul must be a valid Jul pointer.

JulDaysInQuarter(pJul)

JUL - 577:	pJul must be a valid Jul pointer.

JulDaysInYear(pJul)

JUL - 624:	pJul must be a valid Jul pointer.

JulDiff(pJul1, pJul2)

JUL - 663:	pJul1 must be a valid Jul pointer.
JUL - 664:	pJul2 must be a valid Jul pointer.
JUL - 667:	Difference too large for MediumInt

JulDiffL(pJul1, pJul2)

JUL - 703:	pJul1 must be a valid Jul pointer.
JUL - 704:	pJul2 must be a valid Jul pointer.

JulGetSystemJulianDay(pJul)

JUL - 737:	pJul must be a valid Jul pointer.

JulInit(pJul)

None

JulIsLeapYear(yr)

None

JulIsMaxValue(pJul)

JUL - 829:	pJul must be a valid Jul pointer.

JulMax(pJul, pJul1, pJul2)

JUL - 861:	pJul must be a valid Jul pointer.
JUL - 862:	pJul1 must be a valid Jul pointer.
JUL - 863:	pJul2 must be a valid Jul pointer.

JulMin(pJul, pJul1, pJul2)

JUL - 894:	pJul must be a valid Jul pointer.
JUL - 895:	pJul1 must be a valid Jul pointer.
JUL - 896:	pJul2 must be a valid Jul pointer.

JulMonthDayDiff(pJul, mnth, dy)

JUL - 932:	pJul must be a valid Jul pointer.

JulMonthString(pJul, pStr)

JUL - 979:	pJul must be a valid Jul pointer.

JulQuarterString(pJul, pStr)

JUL - 1028:	pJul must be a valid Jul pointer.

JulSameDayMonth(pJul1, pJul2)

JUL - 1073:	pJul1 must be a valid Jul pointer.
JUL - 1074:	pJul2 must be a valid Jul pointer.

JulSetMaxDate(pJul)

JUL - 1107:	pJul must be a valid Jul pointer.

JulToCalendar(pJul, year, month, day)

JUL - 1143:	pJul must be a valid Jul pointer.

JulToDateStr(pJul, pStr, format)

JUL - 1218:	pJul must be a valid Jul pointer.

JulValidateDate(pStr, format)

None

JulWeekString(pJul, pStr)

JUL - 1328:	pJul must be a valid Jul pointer.

JulYearString(pJul, pStr)

JUL - 1375:	pJul must be a valid Jul pointer.

LelAsObj(pLel)

LEL - 73:	pLel must be a valid Lel pointer.

LelClientCount(pLel, offset, pBlk)

LEL - 121:	The list element can be NULL in which case the return value is zero and no items are processed
LEL - 122:	pBlk must be a valid Blk pointer.
LEL - 125:	An invalid ListElement pointer was encountered during the walk

LelClientDll(pLel, offset)

LEL - 160:	pLel must be a valid Lel pointer.

LelClientFindRange(pLelBeg, pLelEnd, offset, pBlk)

LEL - 223:	pLelBeg must be a valid Lel pointer.
LEL - 224:	pLelEnd must be a valid Lel pointer.
LEL - 225:	If pLelBeg does not equal pLelEnd then pLelBeg must precede pLelEnd.
LEL - 226:	pBlk must be a valid Blk pointer.
LEL - 231:	An invalid List Element pointer was encountered during the walk

LelClientNext(pLel, offset)

LEL - 272:	pLel must be a valid Lel pointer.

LelClientPrev(pLel, offset)

LEL - 309:	pLel must be a valid Lel pointer.

LelClientVisitBwd(pLel, offset, pBlk)

LEL - 356:	The list element may be NULL in which case no elements are visited
LEL - 357:	pBlk must be a valid Blk pointer.
LEL - 360:	An invalid ListElement pointer was encountered during the walk

LelClientVisitFwd(pLel, offset, pBlk)

LEL - 408:	The list element may be NULL in which case no elements are visited
LEL - 409:	pBlk must be a valid Blk pointer.
LEL - 412:	An invalid ListElement pointer was encountered during the walk

LelClientVisitPredecessors(pLel, offset, pBlk)

LEL - 459:	pLel must be a valid Lel pointer.
LEL - 460:	pBlk must be a valid Blk pointer.

LelClientVisitRange(pLelBeg, pLelEnd, offset, pBlk)

LEL - 520:	pLelBeg must be a valid Lel pointer.
LEL - 521:	pLelEnd must be a valid Lel pointer.
LEL - 522:	If pLelBeg does not equal pLelEnd then pLelBeg must precede pLelEnd.
LEL - 523:	pBlk must be a valid Blk pointer.
LEL - 529:	An invalid ListElement pointer was encountered during the walk

LelClientVisitSuccessors(pLel, offset, pBlk)

LEL - 577:	pLel must be a valid Lel pointer.
LEL - 578:	pBlk must be a valid Blk pointer.

LelCountRange(pLelBeg, pLelEnd)

LEL - 621:	pLelB must be a valid Lel pointer.
LEL - 622:	pLelE must be a valid Lel pointer.
LEL - 623:	If pLelBeg does not equal pLelEnd then pLelBeg must precede pLelEnd.
LEL - 632:	An invalid ListElement pointer was encountered during the walk

LelCut(pLel)

LEL - 663:	pLel must be a valid Lel pointer.

LelCutRange(pLelBeg, pLelEnd)

LEL - 706:	pLelBeg must be a valid Lel pointer.
LEL - 707:	pLelEnd must be a valid Lel pointer.
LEL - 708:	If pLelBeg does not equal pLelEnd then pLelBeg must precede pLelEnd.

LelCutRangeFromList(pLelBeg, pLelEnd)

LEL - 745:	pLelBeg must be a valid Lel pointer.
LEL - 746:	pLelEnd must be a valid Lel pointer.
LEL - 747:	If pLelBeg does not equal pLelEnd then pLelBeg must precede pLelEnd.
LEL - 756:	An invalid ListElement pointer was encountered while walking through the range

LelDeInit(pLel)

LEL - 790:	pLel must be a valid Lel pointer.
LEL - 791:	The list element must not be connected to any other elements
LEL - 792:	The list element must not be in a list
LEL - 794:	pLel is not a valid ListElement pointer

LelDestroy(pLel)

LEL - 824:	pLel must be a valid Lel pointer.
LEL - 825:	The instance must be the outermost subclass; it cannot have a sub-object

LelGetClient(pLel, offset)

LEL - 857:	pLel must be a valid Lel pointer.

LelGetDll(pLel)

LEL - 887:	pLel must be a valid Lel pointer.

LelGetNext(pLel)

LEL - 917:	pLel must be a valid Lel pointer.

LelGetNthSuccessor(pLel, offset)

LEL - 953:	pLel must be a valid Lel pointer.

LelGetPrev(pLel)

LEL - 985:	pLel must be a valid Lel pointer.

LelInList(pLel)

LEL - 1014:	pLel must be a valid Lel pointer.

LelInit(pLel)

LEL - 1043:	pLel must be a valid Lel pointer.

LelPasteRangeAfter(pLel, pLelBeg, pLelEnd)

LEL - 1090:	pLel must be a valid Lel pointer.
LEL - 1091:	pLelBeg must be a valid Lel pointer.
LEL - 1092:	pLelEnd must be a valid Lel pointer.
LEL - 1093:	If pLelBeg does not equal pLelEnd then pLelBeg must precede pLelEnd.
LEL - 1094:	pLelBeg must not already be in a list.
LEL - 1095:	pLel cannot be identical to either pLelBeg or pLelEnd

LelPasteRangeBefore(pLel, pLelBeg, pLelEnd)

LEL - 1154:	pLel must be a valid Lel pointer.
LEL - 1155:	pLelBeg must be a valid Lel pointer.
LEL - 1156:	pLelEnd must be a valid Lel pointer.
LEL - 1157:	If pLelBeg does not equal pLelEnd then pLelBeg must precede pLelEnd.
LEL - 1158:	pLelBeg must not already be in a list.
LEL - 1159:	pLel cannot be identical to either pLelBeg or pLelEnd

LelPasteRangeToList(pLelBeg, pLelEnd, pDll)

LEL - 1196:	pLelBeg must be a valid Lel pointer.
LEL - 1197:	pLelEnd must be a valid Lel pointer.
LEL - 1198:	If pLelBeg does not equal pLelEnd then pLelBeg must precede pLelEnd.
LEL - 1199:	pLelBeg must not already be in a list.
LEL - 1200:	The list pDll can be NULL
LEL - 1207:	An invalid ListElement pointer was encountered while walking through the range
LEL - 1214:	pDll must be a valid Dll pointer.

LelSendDestroy(pLel)

LEL - 1242:	pLel must be a valid Lel pointer.

LelTest(pLel)

LEL - 1265:	pLel cannot be NULL
LEL - 1266:	pLel must be a valid Lel pointer.
LEL - 1269:	The successor list element must point to pLel
LEL - 1271:	The predecessor list element must point to pLel

LelVisitRange(pLelBeg, pLelEnd, pBlk)

LEL - 1318:	pLelBeg must be a valid Lel pointer.
LEL - 1319:	pLelEnd must be a valid Lel pointer.
LEL - 1320:	If pLelBeg does not equal pLelEnd then pLelBeg must precede pLelEnd.
LEL - 1321:	pBlk must be a valid Blk pointer.
LEL - 1326:	An invalid ListElement pointer was encountered while walking through the range

MclCreateClass(pMcl)

MCL - 105:	pMcl must be a valid Mcl pointer.
MCL - 108:	Not enough memory to create class

MclDestroyClass(pMcl, pCls)

MCL - 146:	pMcl must be a valid Mcl pointer.

Class Exceptions Reference

MclFindFirstSelector(pMcl, pStr)

MCL - 188: pMcl must be a valid Mcl pointer.

MclFindSelector(pMcl, pStr)

MCL - 236: An Mms with a pMth == -1 must be followed by same selector

MclFindSuperClass(pMcl, pStr)

MCL - 271: pMcl must be a valid Mcl pointer.

MclGetClassName(pMcl)

MCL - 303: pMcl must be a valid Mcl pointer.

MclGetClassSize(pMcl)

MCL - 331: pMcl must be a valid Mcl pointer.

MclGetMessageCount(pMcl)

MCL - 359: pMcl must be a valid Mcl pointer.

MclGetNthMms(pMcl, n)

MCL - 389: pMcl must be a valid Mcl pointer.
MCL - 390: The message number must be greater than or equal to zero, and less than the number of messages.

MclGetNthOffset(pMcl, n)

MCL - 425: pMcl must be a valid Mcl pointer.
MCL - 426: The super class index must be greater than or equal to zero, and less than the number of superclasses.

MclGetNthSuper(pMcl, n)

MCL - 456: pMcl must be a valid Mcl pointer.
MCL - 457: The super class index must be greater than or equal to zero, and less than the number of superclasses.

MclGetObjectSize(pMcl)

MCL - 482:	pMcl must be a valid Mcl pointer.

MclGetSuperClassCount(pMcl)

MCL - 508:	pMcl must be a valid Mcl pointer.

MclPrint(pMcl, pCio, item, level, name)

MCL - 544:	pMcl must be a valid Mcl pointer.
MCL - 545:	pCio must be a valid Cio pointer.

MclSendCreateClass(pMcl)

MCL - 586:	pMcl must be a valid Mcl pointer.
MCL - 593:	pMcl failed to create the class

MclSendDestroyClass(pMcl, pCls)

MCL - 617:	pMcl must be a valid Mcl pointer.

MclValidate(pMcl)

MCL - 650:	pMcl must be a valid Mcl pointer.
MCL - 652:	The name component must not be NULL
MCL - 653:	The number of messages must be in the range [0:100]
MCL - 654:	The number of superclasses must be in the range [0:30]
MCL - 657:	A create function must be supplied
MCL - 658:	A destroy function must be supplied

MclValidateMessages(pMcl)

MCL - 690:	pMcl must be a valid Mcl pointer.

MclValidateSuperClasses(pMcl)

MCL - 726:	pMcl must be a valid Mcl pointer.

MemClear(pMem, s, i, n, os)

MEM - 86:	pMem must be a valid Mem pointer.
MEM - 87:	i must be less s.
MEM - 88:	(i + n) must be less or equal to s.
MEM - 89:	os must be greater than 0.

MemCopy(pMem, s, di, n, i, os)

MEM - 128:	pMem must be a valid Mem pointer.
MEM - 129:	i must be less s.
MEM - 130:	(i + n) must be less or equal to s.
MEM - 131:	os must be greater than 0.
MEM - 132:	di must be less s.
MEM - 133:	(di + n) must be less or equal to s.

MemCutNSet(pMem, s, i, n, os, c)

MEM - 174:	pMem must be a valid Mem pointer.
MEM - 175:	i must be less s.
MEM - 176:	(i + n) must be less or equal to s.
MEM - 177:	os must be greater than 0.

MemDestroy(pMem)

MEM - 216:	pMem must be a valid Mem pointer.

MemDuplicate(pMem, s, pMemS, os)

MEM - 246:	pMem must be a valid Mem pointer.
MEM - 247:	pMemS must be a valid Mem pointer.
MEM - 248:	os must be greater than 0.

MemNew(amount)

None

MemNewFast(amount)

None

MemPasteNSet(pMem, s, i, n, os, c)

MEM - 363:	pMem must be a valid Mem pointer.
MEM - 364:	i must be less s.
MEM - 365:	(i + n) must be less or equal to s.
MEM - 366:	os must be greater than 0.

MemSetChr(pMem, s, i, n, os, c)

MEM - 403:	pMem must be a valid Mem pointer.
MEM - 404:	i must be less s.
MEM - 405:	(i + n) must be less or equal to s.
MEM - 406:	os must be greater than 0.

MmsGetMethod(pMms)

MMS - 88:	pMms must be a valid Mms pointer.

MmsGetSuper(pMms)

MMS - 144:	pMms must be a valid Mms pointer.

MmsGetSelector(pMms)

MMS - 181:	pMms must be a valid Mms pointer.

MmsPrint(pMms, pCio, item, level, name)

MMS - 214:	pMms must be a valid Mms pointer.
MMS - 215:	pCio must be a valid Cio pointer.

MmsValidate(pMms)

MMS - 246:	pMms must be a valid Mms pointer.
MMS - 248:	pMms->selector must be a valid Str pointer.

MscGetName(pMsc)

MSC - 79:	pMsc must be a valid Msc pointer.

Class Exceptions Reference

MscGetMetaClass(pMsc)

MSC - 109:	pMsc must be a valid Msc pointer.

MscGetOffset(pMsc)

MSC - 143:	pMsc must be a valid Msc pointer.
MSC - 146:	The superobject offset must be in the range [0:2000]

MscPrint(pMsc, pCio, item, level, name)

MSC - 178:	pMsc must be a valid Msc pointer.
MSC - 179:	pCio must be a valid Cio pointer.

MscValidate(pMsc)

MSC - 215:	pMsc must be a valid Msc pointer.
MSC - 217:	A name must be supplied
MSC - 218:	A valid MetaClas pointer must be supplied
MSC - 219:	A pointer to an instance of the super-object must be supplied
MSC - 220:	A pointer to an instance of the super-object must be supplied
MSC - 221:	A pointer to a MediumInt must be supplied

MsgDeInit(pMsg)

MSG - 80:	pMsg must be a valid Msg pointer.
MSG - 81:	pMsg must be a valid Msg pointer.

MsgGetOffset(pMsg)

MSG - 106:	pMsg must be a valid Msg pointer.

MsgGetMethod(pMsg)

MSG - 133:	pMsg must be a valid Msg pointer.

MsgGetSelector(pMsg)

MSG - 160:	pMsg must be a valid Msg pointer.

MsgInit(pMsg, pCls, pMms)

MSG - 190:	pCls must be a valid Cls pointer.
MSG - 191:	pMms must be a valid Mms pointer.
MSG - 192:	pMsg must be a valid Msg pointer.
MSG - 203:	The superclass name referred to by pMms is not a superclass of pCls
MSG - 205:	The selector is invalid for the superclass named by pMms
MSG - 208:	pMms cannot override the selector named
MSG - 212:	pMms cannot inherit the selector named

MsgPrint(pMsg, pCio, item, level, name)

MSG - 249:	pMsg must be a valid Msg pointer.
MSG - 250:	pCio must be a valid Cio pointer.

MsgSend(pMsg, pObj, pBlk)

MSG - 286:	pMsg must be a valid Msg pointer.
MSG - 287:	pBlk must be a valid Blk pointer.

MsgSendReturnInt(pMsg, pObj, pBlk)

MSG - 319:	pMsg must be a valid Msg pointer.
MSG - 320:	pBlk must be a valid Blk pointer.

MsgSendReturnPtr(pMsg, pObj, pBlk)

MSG - 352:	pMsg must be a valid Msg pointer.
MSG - 353:	pBlk must be a valid Blk pointer.

MsgSetSuperOffsetAndMethod(pMsg, pMsgSuper)

MSG - 382:	pMsg must be a valid Msg pointer.
MSG - 383:	pMsgSuper must be a valid Msg pointer.

ObjDeInit(pObj)

OBJ - 83:	pObj must be a valid Obj pointer.
OBJ - 85:	pObj must be a valid Obj pointer.

Class Exceptions Reference

ObjDestroy(pObj)

OBJ - 116:	pObj must be a valid Obj pointer.

ObjGetClient(pObj, offset)

OBJ - 155:	pObj must be a valid Obj pointer.

ObjGetClientOrNull(pObj, offset)

OBJ - 198:	pObj may be NULL

ObjGetCls(pObj)

OBJ - 225:	pObj must be a valid Obj pointer.

ObjGetClsName(pObj)

OBJ - 252:	pObj must be a valid Obj pointer.

ObjGetImmediateClient(pObj)

OBJ - 282:	pObj must be a valid Obj pointer.

ObjGetMethodAndOffset(pObj, m, ppMth, pOffset)

OBJ - 324:	pObj must be a valid Obj pointer.
OBJ - 325:	The message number must be greater than or equal to zero, and less than the number of messages.

ObjGetNthSuperObject(pObj, n)

OBJ - 354:	pObj must be a valid Obj pointer.
OBJ - 355:	The super class index must be greater than or equal to zero, and less than the number of superclasses.

ObjGetRootClient(pObj)

OBJ - 386:	pObj must be a valid Obj pointer.

ObjGetRootClientSize(pObj)

OBJ - 415:	pObj must be a valid Obj pointer.

ObjGetSize(pObj)

OBJ - 442:	pObj must be a valid Obj pointer.

ObjGetSubObjectOffset(pObj)

OBJ - 471:	pObj must be a valid Obj pointer.

ObjInit(pObj, pCls)

OBJ - 506:	pCls must be a valid Cls pointer.
OBJ - 507:	pObj must be a valid Obj pointer.

ObjIsRoot(pObj)

OBJ - 536:	pObj must be a valid Obj pointer.

ObjRespondsToSelector(pObj, pStr)

OBJ - 567:	pObj must be a valid Obj pointer.

ObjSendMessage(pObj, m, pBlk)

OBJ - 598:	pObj must be a valid Obj pointer.
OBJ - 599:	The message number must be greater than or equal to zero, and less than the number of messages.
OBJ - 600:	pBlk must be a valid Blk pointer.

ObjSendMessageReturnInt(pObj, m, pBlk)

OBJ - 635:	pObj must be a valid Obj pointer.
OBJ - 636:	The message number must be greater than or equal to zero, and less than the number of messages.
OBJ - 637:	pBlk must be a valid Blk pointer.

Class Exceptions Reference

ObjSendMessageReturnPtr(pObj, m, pBlk)

OBJ - 672:	pObj must be a valid Obj pointer.
OBJ - 673:	The message number must be greater than or equal to zero, and less than the number of messages.
OBJ - 674:	pBlk must be a valid Blk pointer.

StrBasicExtract(pStrD, pStrS, index, length)

STR - 62:	pStrD must be a valid Str pointer.
STR - 63:	pStrS must be a valid Str pointer.

StrExtract(pStrD, pStrS, idx, n)

STR - 111:	pStrD must be a valid Str pointer.
STR - 112:	pStrS must be a valid Str pointer.

StrFromDate(pStr, format, year, month, day)

STR - 165:	pStr must be a valid Str pointer.
STR - 178:	Invalid format

StrFromMediumInt(pStr, maxLen, num)

STR - 215:	pStr must be a valid Str pointer.

StrInit(pStr)

STR - 248:	pStr must be a valid Str pointer.

StrReplaceSubStr(pStrOrg, pStrFrom, pStrTo, maxLen)

STR - 285:	pStrOrg must be a valid Str pointer.
STR - 286:	pStrFrom must be a valid Str pointer.
STR - 287:	pStrTo must be a valid Str pointer.
STR - 288:	maxLen must be in the range [0:128]

StrSet(pStrD, pStrS)

STR - 329:	pStrD must be a valid Str pointer.
STR - 330:	pStrS must be a valid Str pointer.

StrSqueeze(pStrD, pStrS, ch)

STR - 360:	s must be a valid Str pointer.
STR - 361:	pStrD must be a valid Str pointer.

StrToDate(pStr, format, year, month, day)

STR - 412:	pStr must be a valid Str pointer.
STR - 413:	The length of pStr must be less than 12
STR - 427:	Invalid format

StrToLower(pStr)

STR - 452:	pStr must be a valid Str pointer.

StrToMediumInt(pStr)

STR - 483:	pStr must be a valid Str pointer.

StrToUpper(pStr)

STR - 509:	pStr must be a valid Str pointer.

TreAsDll(pTre)

TRE - 80:	pTre must be a valid Tre pointer.

TreAsLel(pTre)

TRE - 113:	pTre must be a valid Tre pointer.

TreAsObj(pTre)

None

TreClear(pTre)

TRE - 168:	pTre must be a valid Tre pointer.
TRE - 172:	pTre must have no child nodes.
TRE - 177:	pTre must be the root node.

Class Exceptions Reference

TreClient(pTre, offset)

TRE - 214: pTre must be a valid Tre pointer.

TreClientFindChild(pTre, offset, pBlk)

TRE - 266: pTre must be a valid Tre pointer.

TreClientFirstChild(pTre, offset)

TRE - 307: pTre must be a valid Tre pointer.

TreClientLastChild(pTre, offset)

TRE - 348: pTre must be a valid Tre pointer.

TreClientLastLeaf(pTre, offset)

TRE - 391: pTre must be a valid Tre pointer.

TreClientNext(pTre, offset)

TRE - 432: pTre must be a valid Tre pointer.

TreClientNextPreOrder(pTre, offset)

TRE - 475: pTre must be a valid Tre pointer.

TreClientNextUncle(pTre, offset)

TRE - 519: pTre must be a valid Tre pointer.

TreClientParent(pTre, offset)

TRE - 559: pTre must be a valid Tre pointer.

TreClientPrev(pTre, offset)

TRE - 600:	pTre must be a valid Tre pointer.

TreClientPrevPreOrder(pTre, offset)

TRE - 643:	pTre must be a valid Tre pointer.

TreClientVisitBranchInOrder(pTre, offset, pBlk)

TRE - 695:	pTre must be a valid Tre pointer.
TRE - 696:	pBlk must be a valid Blk pointer.

TreClientVisitChildren(pTre, offset, pBlk)

TRE - 751:	pTre must be a valid Tre pointer.
TRE - 752:	pBlk must be a valid Blk pointer.

TreClientVisitChildrenBwd(pTre, offset, pBlk)

TRE - 805:	pTre must be a valid Tre pointer.
TRE - 806:	pBlk must be a valid Blk pointer.

TreClientVisitDescBranchInOrder(pTre, offset, pBlk)

TRE - 862:	pTre must be a valid Tre pointer.
TRE - 863:	pBlk must be a valid Blk pointer.

TreClientVisitDescInOrder(pTre, offset, pBlk)

TRE - 922:	pTre must be a valid Tre pointer.
TRE - 923:	pBlk must be a valid Blk pointer.

TreClientVisitDescInOrderBwd(pTre, offset, pBlk)

TRE - 982:	pTre must be a valid Tre pointer.
TRE - 983:	pBlk must be a valid Blk pointer.

TreClientVisitDescLeaves(pTre, offset, pBlk)

TRE - 1041:	pTre must be a valid Tre pointer.
TRE - 1042:	pBlk must be a valid Blk pointer.

TreClientVisitDescPreOrder(pTre, offset, pBlk)

TRE - 1100:	pTre must be a valid Tre pointer.
TRE - 1101:	pBlk must be a valid Blk pointer.

TreClientVisitInOrder(pTre, offset, pBlk)

TRE - 1157:	pTre must be a valid Tre pointer.
TRE - 1158:	pBlk must be a valid Blk pointer.

TreClientVisitInOrderBwd(pTre, offset, pBlk)

TRE - 1212:	pTre must be a valid Tre pointer.
TRE - 1213:	pBlk must be a valid Blk pointer.

TreClientVisitLeaves(pTre, offset, pBlk)

TRE - 1266:	pTre must be a valid Tre pointer.
TRE - 1267:	pBlk must be a valid Blk pointer.

TreClientVisitParents(pTre, offset, pBlk)

TRE - 1323:	pTre must be a valid Tre pointer.
TRE - 1324:	pBlk must be a valid Blk pointer.

TreClientVisitPreOrder(pTre, offset, pBlk)

TRE - 1379:	pTre must be a valid Tre pointer.
TRE - 1380:	pBlk must be a valid Blk pointer.

TreClientVisitRange(pTreBeg, pTreEnd, offset, pBlk)

TRE - 1445:	pTreBeg must be a valid Tre pointer.
TRE - 1446:	pTreEnd must be a valid Tre pointer.
TRE - 1447:	If pTreBeg does not equal pTreEnd then pTreBeg must precede pTreEnd.
TRE - 1448:	pBlk must be a valid Blk pointer.

TreClientVisitSuccPreOrder(pTre, offset, pBlk)

TRE - 1503:	pTre must be a valid Tre pointer.
TRE - 1504:	pBlk must be a valid Blk pointer.
TRE - 1508:	An invalid Tree pointer was encountered while walking the tree

TreClientVisitSuccessors(pTre, offset, pBlk)

TRE - 1561:	pTre must be a valid Tre pointer.
TRE - 1562:	pBlk must be a valid Blk pointer.

TreCutChildren(pTre)

TRE - 1591:	pTre must be a valid Tre pointer.

TreCutRange(pTreBeg, pTreEnd)

TRE - 1632:	pTreBeg must be a valid Tre pointer.
TRE - 1633:	pTreEnd must be a valid Tre pointer.
TRE - 1634:	If pTreBeg does not equal pTreEnd then pTreBeg must precede pTreEnd.

TreDeInit(pTre)

TRE - 1663:	pTre must be a valid Tre pointer.
TRE - 1664:	pTre must be the root node.
TRE - 1665:	pTre must have no child nodes.
TRE - 1669:	pTre must be a valid Tre pointer.

TreDestroy(pTre)

TRE - 1697:	pTre must be a valid Tre pointer.
TRE - 1698:	pTre must not have a sub-object

TreDestroyChildren(pTre)

TRE - 1730:	pTre must be a valid Tre pointer.

TreFirstChild(pTre)

TRE - 1764:	pTre must be a valid Tre pointer.

TreHasChildren(pTre)

TRE - 1794:	pTre must be a valid Tre pointer.

TreHasSiblings(pTre)

TRE - 1825:	pTre must be a valid Tre pointer.

TreInit(pTre)

TRE - 1854:	pTre must be a valid Tre pointer.

TreIsChild(pTre)

TRE - 1884:	pTre must be a valid Tre pointer.

TreIsDirectAncestor(pTre, pTreA)

TRE - 1920:	pTre must be a valid Tre pointer.
TRE - 1921:	pTreA must be a valid Tre pointer.

TreIsRoot(pTre)

TRE - 1957:	pTre must be a valid Tre pointer.

TreLastChild(pTre)

TRE - 1990:	pTre must be a valid Tre pointer.

TreLastLeaf(pTre)

TRE - 2026:	pTre must be a valid Tre pointer.
TRE - 2031:	An invalid Tree pointer was encountered while walking the tree

TreNext(pTre)

TRE - 2063:	pTre must be a valid Tre pointer.

TreNextPreOrder(pTre)

TRE - 2097:	pTre must be a valid Tre pointer.

TreNextUncle(pTre)

TRE - 2138:	pTre must be a valid Tre pointer.
TRE - 2142:	An invalid Tree pointer was encountered while walking the tree

TreParent(pTre)

TRE - 2178:	pTre must be a valid Tre pointer.

TrePasteRangeAfterSibling(pTre, pTreBeg, pTreEnd)

TRE - 2226:	pTre must be a valid Tre pointer.
TRE - 2227:	pTre must be a parent.
TRE - 2228:	pTreBeg must be a valid Tre pointer.
TRE - 2229:	pTreEnd must be a valid Tre pointer.
TRE - 2230:	If pTreBeg does not equal pTreEnd then pTreBeg must precede pTreEnd.
TRE - 2231:	pTreBeg must be the root node.

TrePasteRangeBeforeSibling(pTre, pTreBeg, pTreEnd)

TRE - 2279:	pTre must be a valid Tre pointer.
TRE - 2280:	pTre must be a parent.
TRE - 2281:	pTreBeg must be a valid Tre pointer.
TRE - 2282:	pTreEnd must be a valid Tre pointer.
TRE - 2283:	If pTreBeg does not equal pTreEnd then pTreBeg must precede pTreEnd.
TRE - 2284:	pTreBeg must be the root node.

TrePasteRangeFirstChild(pTre, pTreBeg, pTreEnd)

TRE - 2331:	pTre must be a valid Tre pointer.
TRE - 2332:	pTreBeg must be a valid Tre pointer.
TRE - 2333:	pTreEnd must be a valid Tre pointer.
TRE - 2334:	If pTreBeg does not equal pTreEnd then pTreBeg must precede pTreEnd.
TRE - 2335:	pTreBeg must be the root node.

TrePasteRangeLastChild(pTre, pTreBeg, pTreEnd)

TRE - 2382:	pTre must be a valid Tre pointer.
TRE - 2383:	pTreBeg must be a valid Tre pointer.
TRE - 2384:	pTreEnd must be a valid Tre pointer.
TRE - 2385:	If pTreBeg does not equal pTreEnd then pTreBeg must precede pTreEnd.
TRE - 2386:	pTreBeg must be the root node.

TrePrev(pTre)

TRE - 2419:	pTre must be a valid Tre pointer.

TrePrevPreOrder(pTre)

TRE - 2456:	pTre must be a valid Tre pointer.

TreSendDestroy(pTre)

TRE - 2488:	pTre must be a valid Tre pointer.

TreVisitBranchInOrder(pTre, pBlk)

TRE - 2535:	pTre must be a valid Tre pointer.
TRE - 2536:	pBlk must be a valid Blk pointer.

TreVisitChildren(pTre, pBlk)

TRE - 2581:	pTre must be a valid Tre pointer.
TRE - 2582:	pBlk must be a valid Blk pointer.

TreVisitChildrenBwd(pTre, pBlk)

TRE - 2627:	pTre must be a valid Tre pointer.
TRE - 2628:	pBlk must be a valid Blk pointer.

TreVisitDescBranchInOrder(pTre, pBlk)

TRE - 2674:	pTre must be a valid Tre pointer.
TRE - 2675:	pBlk must be a valid Blk pointer.

TreVisitDescInOrder(pTre, pBlk)

TRE - 2720:	pTre must be a valid Tre pointer.
TRE - 2721:	pBlk must be a valid Blk pointer.

TreVisitDescInOrderBwd(pTre, pBlk)

TRE - 2766:	pTre must be a valid Tre pointer.
TRE - 2767:	pBlk must be a valid Blk pointer.

TreVisitDescPreOrder(pTre, pBlk)

TRE - 2811:	pTre must be a valid Tre pointer.
TRE - 2812:	pBlk must be a valid Blk pointer.

TreVisitInOrder(pTre, pBlk)

TRE - 2857:	pTre must be a valid Tre pointer.
TRE - 2858:	pBlk must be a valid Blk pointer.

TreVisitInOrderBwd(pTre, pBlk)

TRE - 2903:	pTre must be a valid Tre pointer.
TRE - 2904:	pBlk must be a valid Blk pointer.

TreVisitLeaves(pTre, pBlk)

TRE - 2948:	pTre must be a valid Tre pointer.
TRE - 2949:	pBlk must be a valid Blk pointer.

TreVisitParents(pTre, pBlk)

TRE - 2992:	pTre must be a valid Tre pointer.
TRE - 2993:	pBlk must be a valid Blk pointer.

TreVisitPreOrder(pTre, pBlk)

TRE - 3038:	pTre must be a valid Tre pointer.
TRE - 3039:	pBlk must be a valid Blk pointer.

TreVisitRange(pTreBeg, pTreEnd, pBlk)

TRE - 3093:	pTreBeg must be a valid Tre pointer.
TRE - 3094:	pTreEnd must be a valid Tre pointer.
TRE - 3095:	If pTreBeg does not equal pTreEnd then pTreBeg must precede pTreEnd.
TRE - 3096:	pBlk must be a valid Blk pointer.

TreVisitSuccPreOrder(pTre, pBlk)

TRE - 3140:	pTre must be a valid Tre pointer.
TRE - 3141:	pBlk must be a valid Blk pointer.

TreVisitSuccessors(pTre, pBlk)

TRE - 3186:	pTre must be a valid Tre pointer.
TRE - 3187:	pBlk must be a valid Blk pointer.

TskBasicAssert(condition, fileName, lineNo)

None

TskCondition(pTsk, type, error, fileName, lineNo)

TSK - 122:	pTsk must be a valid Tsk pointer.
TSK - 123:	The exception stack must not be empty for the Task pTsk.
TSK - 124:	There must be no recursion of the exception.
TSK - 125:	The condition category must be valid.

TskDeInit(pTsk)

TSK - 159:	pTsk must be a valid Tsk pointer.
TSK - 163:	pTsk must be a valid Tsk pointer.

TskDefaultInitDos(pTsk, argc, argv)

None

TskExit(pTsk, exitVal)

TSK - 241:	pTsk must be a valid Tsk pointer.

TskExitWithMsg(pTsk, msg)

TSK - 288:	pTsk must be a valid Tsk pointer.

TskGetArgc(pTsk)

TSK - 315:	pTsk must be a valid Tsk pointer.

TskGetArgv(pTsk)

TSK - 345:	pTsk must be a valid Tsk pointer.

TskGetExceptionCondition(pTsk)

TSK - 376:	pTsk must be a valid Tsk pointer.

TskGetExceptionFileName(pTsk)

TSK - 408:	pTsk must be a valid Tsk pointer.

TskGetExceptionLineNo(pTsk)

TSK - 439:	pTsk must be a valid Tsk pointer.

TskGetExceptionType(pTsk)

TSK - 468:	pTsk must be a valid Tsk pointer.

TskInitDos(pTsk, argc, argv, maxNesting, excFilter)

TSK - 518: pTsk must be a valid Tsk pointer.

TskIsInitialized(Void)

None

TskLogCond(pTsk, error)

None

TskMainLogCond(error)

None

TskMainPreCond(error)

None

TskMainPtrCond(error)

None

TskMainRaiseException(error)

None

TskNormalExit(pTsk)

TSK - 776: pTsk must be a valid Tsk pointer.

TskOnException(pTsk)

None

TskPopExceptionHandler(pTsk)

TSK - 854:	pTsk must be a valid Tsk pointer.
TSK - 855:	The exception stack must not be empty for the Task pTsk.

TskPreCond(pTsk, error)

None

TskPrintException(pTsk)

TSK - 918:	pTsk must be a valid Tsk pointer.

TskPropagateException(pTsk, useFilter)

TSK - 952:	pTsk must be a valid Tsk pointer.
TSK - 953:	The exception stack must not be empty for the Task pTsk.
TSK - 954:	There must be no recursion of the exception.

TskPtrCond(pTsk, error)

None

TskPushExh(pTsk)

TSK - 1025:	pTsk must be a valid Tsk pointer.
TSK - 1026:	The exception stack must not be full for the Task pTsk.

TskRaiseException(pTsk, error)

None

VtxAsGrfLel(pVtx)

VTX - 95:	pVtx must be a valid Vtx pointer.

VtxAsInDll(pVtx)

VTX - 127:	pVtx must be a valid Vtx pointer.

VtxAsObj(pVtx)

VTX - 160:	pVtx must be a valid Vtx pointer.

VtxAsOutDll(pVtx)

VTX - 192:	pVtx must be a valid Vtx pointer.

VtxClear(pVtx)

VTX - 219:	pVtx must be a valid Vtx pointer.
VTX - 225:	Failed to disconnect from edges
VTX - 229:	pVtx cannot have edges without being in a Graph

VtxConnectToGrf(pVtx, pGrf)

VTX - 257:	pVtx must be a valid Vtx pointer.
VTX - 258:	pGrf must be a valid Grf pointer.
VTX - 259:	pVtx must not already be linked to a graph.
VTX - 260:	pVtx must have no edges.

VtxCountIn(pVtx)

VTX - 291:	pVtx must be a valid Vtx pointer.

VtxCountOut(pVtx)

VTX - 322:	pVtx must be a valid Vtx pointer.

VtxDeInit(pVtx)

VTX - 354:	pVtx must be a valid Vtx pointer.
VTX - 355:	pVtx must have no edges.
VTX - 356:	pVtx must not already be linked to a graph.
VTX - 362:	pVtx must be a valid Vtx pointer.

VtxDestroy(pVtx)

VTX - 390:	pVtx must be a valid Vtx pointer.
VTX - 391:	pVtx must not have a sub-object

VtxDisconnectFromGrf(pVtx)

VTX - 421:	pVtx must be a valid Vtx pointer.
VTX - 422:	pVtx must be linked to a graph.
VTX - 423:	pVtx must have no edges.

VtxFindOutEdg(pVtxO, pVtxI, offset, pBlk)

VTX - 490:	pVtxO must be a valid Vtx pointer.
VTX - 491:	pVtxI must be a valid Vtx pointer.
VTX - 492:	pBlk must be a valid Blk pointer.

VtxFindOutEdgClient(pVtxO, pVtxI, offset, pBlk)

VTX - 563:	pVtxO must be a valid Vtx pointer.
VTX - 564:	pVtxI must be a valid Vtx pointer.
VTX - 565:	pBlk must be a valid Blk pointer.

VtxGetClient(pVtx, offset)

VTX - 599:	pVtx must be a valid Vtx pointer.

VtxGetFirstIn(pVtx)

VTX - 631:	pVtx must be a valid Vtx pointer.

VtxGetFirstOut(pVtx)

VTX - 663:	pVtx must be a valid Vtx pointer.

VtxGetGrf(pVtx)

VTX - 692:	pVtx must be a valid Vtx pointer.

VtxInit(pVtx)

VTX - 721:	pVtx must be a valid Vtx pointer.

VtxInGrf(pVtx)

VTX - 751:	pVtx must be a valid Vtx pointer.

VtxSendDestroy(pVtx)

VTX - 780:	pVtx must be a valid Vtx pointer.

VtxStackSetup(pVtx, inStack, outStack, stackVtx, i)

VTX - 809:	pVtx must be a valid Vtx pointer.

VtxVisitEdge(pVtx, pBlk)

VTX - 847:	pVtx must be a valid Vtx pointer.
VTX - 848:	pBlk must be a valid Blk pointer.

VtxVisitEdgeClient(pVtx, offset, pBlk)

VTX - 889:	pVtx must be a valid Vtx pointer.
VTX - 890:	pBlk must be a valid Blk pointer.

VtxVisitInEdge(pVtx, pBlk)

VTX - 926:	pVtx must be a valid Vtx pointer.
VTX - 927:	pBlk must be a valid Blk pointer.

VtxVisitInEdgeClient(pVtx, offset, pBlk)

VTX - 969:	pVtx must be a valid Vtx pointer.
VTX - 970:	pBlk must be a valid Blk pointer.

VtxVisitOutEdge(pVtx, pBlk)

VTX - 1006:	pVtx must be a valid Vtx pointer.

VtxVisitOutEdgeClient(pVtx, offset, pBlk)

VTX - 1048:	pVtx must be a valid Vtx pointer.